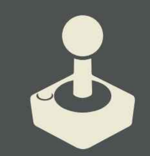
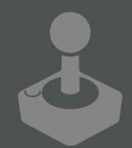



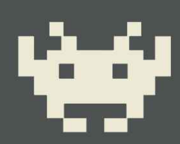


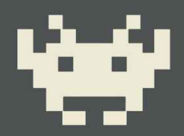


# A PROGRAMAR SE APRENDE JUGANDO

José M.ª Maestre Torreblanca
Jesús Relinque Pérez

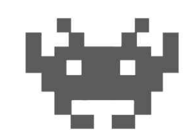






Paraninfo

# A PROGRAMAR SE APRENDE JUGANDO





# A PROGRAMAR SE APRENDE JUGANDO

José M.ª Maestre Torreblanca
Jesús Relinque Pérez





Paraninfo

Paraninfo

**A programar se aprende jugando**


**Gerente Editorial**
María José López Raso

**Equipo Técnico Editorial**
Alicia Cerviño González
Paola Paz Otero

**Editora de Adquisiciones**
Carmen Lara Carmona

**Producción**
Nacho Cabal Ramos

**Diseño de cubierta**
Ediciones Nobel

**Preimpresión**
José M.ª Maestre Torreblanca
Jesús Relinque Pérez

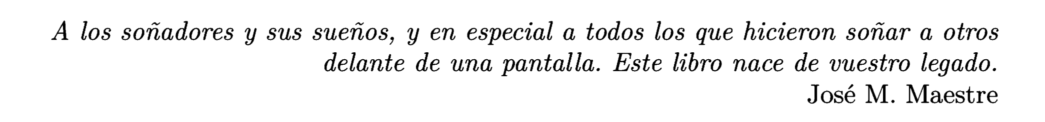

1.ª edición, 2017

C/ Velázquez 31, 3.° Dcha. / 28001 Madrid, ESPAÑA
Teléfono: 902 995 240 / Fax: 914 456 218
clientes@paraninfo.es / www.paraninfo.es

ISBN: 978-84-283-3727-4
Depósito legal: M-16294-2017

(15577)

**Impreso en España / *Printed in Spain***
Cimapress (Arganda del Rey, Madrid)



*A los soñadores y sus sueños, y en especial a todos los que hicieron soñar a otros delante de una pantalla. Este libro nace de vuestro legado.*

José M. Maestre

*Querría dedicar este libro a todo aquel que sintió en alguna ocasión la inquietud de hacer magia mediante un ordenador, transformando un puñado de líneas de código en un videojuego, como si de un truco de prestidigitador se tratara, cuyo esencial cometido fuera evadir la mente de aquel que lo presenciara.*

Jesús Relinque

# Prólogo

Nadie con menos de veinticinco años puede entender completamente la transformación que los ordenadores y la tecnología han obrado en nuestras vidas a lo largo de las últimas décadas. Llamar desde una cabina telefónica, desplegar un arrugado mapa de papel dentro del coche para escoger la mejor ruta o comprar un carrete para la cámara de fotos eran acciones comunes hace apenas treinta años. Hoy son consideradas, en el mejor de los casos, actividades *vintage*. La propia manera de utilizar la tecnología ha cambiado profundamente también para penetrar en nuestro día a día. Controlar a nivel de usuario un ordenador ha dejado de ser el reto que fue antaño, especialmente para los nativos digitales, jóvenes que han crecido sumergidos en un océano tecnológico. Sin embargo, y pese a que tareas como editar un archivo de texto resultan hoy triviales, hay pocas personas que sepan cómo hablar con una máquina, cómo comunicarse con ella para que obedezca y ponga a nuestra disposición sus recursos. Esta habilidad se conoce como programación, y es tan útil que muchas empresas pagan una cantidad considerable de dinero por llevarla a cabo.

### Pura magia

Programar consiste en dar órdenes a una máquina para que las ejecute de forma automática. Exige, únicamente, hablar un idioma que la máquina entienda, un lenguaje de programación. Con las herramientas apropiadas, basta escribir unas decenas de palabras para que nuestros deseos cobren vida en los circuitos de un ordenador. Es magia. Y está al alcance de todo aquel dispuesto a invertir unas horas de su tiempo entre estas páginas.

### Old but gold

El lenguaje de programación que aprenderá en este libro es C, el clásico por antonomasia. Si la música de los *Beatles* o los *Rolling Stones* se convirtiera en código, lo haría en C. Sin duda. Desde su creación hace más de cuarenta años este lenguaje ha permanecido siempre entre los más utilizados y su impronta ha sido tan grande que muchos otros lenguajes están inspirados en él. De hecho, hoy vive una segunda juventud gracias a plataformas de desarrollo como Arduino, que también lo utilizan como lenguaje de programación de base.

La elección de C es también óptima desde el punto de vista didáctico, pues se trata de un lenguaje equilibrado en su dificultad y posibilidades, permitiendo al programador acceder a los secretos mejor guardados de la máquina con poco esfuerzo.

**Bienvenidos a Hogwarts**

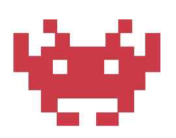
Completar este libro es el primer paso para convertirse en un mago de la programación. Como alumnos de una escuela de magia, aprenderéis primero a realizar programas y juegos sencillos para continuar con retos cada vez más complejos en cada capítulo. A lo largo del libro encontraréis juegos de vuestra niñez y también otros que causaron furor en el pasado. Antes de romper la lista de ventas con un nuevo *Final Fantasy* o crear de una vez por todas el sucesor de *Street Fighter II*, debéis aprender las esencias de la programación. Para ayudar a ello, se presentan cuadros de diferentes colores e iconos:

**Cuadro de palabras mágicas de programación**



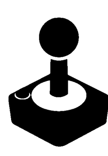
Los cuadros de color azul e iconos de gorro de Merlín se utilizan para presentar palabras mágicas del lenguaje C que el lector deberá recordar el futuro. Será con estas palabras con las que hagamos que el ordenador cobre vida.

**Cuadro de ejemplos de programación**



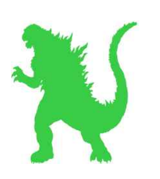
Los cuadros de color rojo e iconos de invasión alienígena se utilizan para presentar ejemplos de programación. Se tratará de trozos sencillos de código encaminados a ilustrar lo que se está explicando.

**Cuadro de juegos**



Los cuadros de color negro e iconos de *joystick* se utilizan para presentar videojuegos realizados en C basados en el contenido del capítulo.

**Cuadro de ejemplos de programación**



Los cuadros de color verde e iconos de Godzilla presentan retos de programación para que pongáis a prueba vuestros conocimientos.

### ¿Por qué videojuegos?

Este es un libro de iniciación a la programación. A diferencia de otros textos, se ha apostado por la programación de videojuegos como vehículo principal de aprendizaje. Gracias a ello, se pueden adquirir de forma amena los conocimientos necesarios para realizar programas sencillos y afrontar sin miedo el aprendizaje de nuevos lenguajes en el futuro. Además, las habilidades del programador son tan transversales que sirven por igual para todo tipo de aplicaciones, desde automatizar la gestión de nóminas en una empresa a trabajar en la próxima entrega de *Mario Kart* o *Tomb Raider*.

### ¿Cómo usar el libro?

Los contenidos de la obra cubren los conocimientos esenciales que se adquieren en un primer curso de programación a nivel universitario, aunque se han introducido algunas simplificaciones con fines didácticos. En total, se presentan un total de siete capítulos que cubren los siguientes temas:

- Introducción a los computadores (capítulo 1): se proporcionan conocimientos básicos de la estructura del computador y de C, aunque su lectura no es imprescindible. Los más impacientes pueden ir directamente al capítulo 2.
- Curso básico de programación en C (capítulos 2 a 6): este bloque explica los conocimientos fundamentales para iniciarse en el mundo de la programación.
- Pinceladas de programación avanzada (Capítulo 7): para concluir, se introducen algunos de los temas que esperan a todo aquel que continúe por la senda de la programación.

### A programar se aprende jugando

A lo largo del libro os presentaremos numerosos ejemplos y también juegos que os ayudarán a convertiros en magos de la programación, pero nada será tan efectivo como que os dejéis llevar por la imaginación. Programad vuestros propios juegos, pensad en cómo ampliarlos y mejorarlos, disfrutadlos y pasadlo bien. A través del juego encontraréis la enorme verdad que se esconde tras una vieja máxima de la programación: "a programar se aprende programando". Con ganas e imaginación aprenderéis jugando.

### Agradecimientos

En primer lugar, agradecerte a ti, lector, por tener este libro en tus manos. Esperamos de corazón que disfrutes el libro y los juegos contenidos en él tanto como nosotros. Deseamos con todas nuestras fuerzas que dichos juegos sirvan de inspiración para tus futuras creaciones dentro del maravilloso universo del ocio electrónico.

Acto seguido, queremos recordar la relevancia de antiguos textos que, allá por los años ochenta, simbolizaban un auténtico *rara avis* en la biblioteca de turno, puesto que servían de material bibliográfico para aprender los fundamentos de la informática. En aquella época resultaba muy complicado encontrar libros en castellano que trataran ese tema, y volúmenes como los que componían la *Colección Electrónica* de

Ediciones Plesa se convirtieron en un verdadero grial para los que nos propusimos ser autodidactas de la programación.

Nos parece de justicia mencionar al profesorado universitario que nos guio por la senda de la algoritmia en la vieja y añorada Escuela Superior de Ingeniería de Cádiz y en la Escuela Superior de Ingeniería de Sevilla. Tampoco podemos olvidar a personas como Alberto Urbano y Pepe Expósito, que con su dedicación y apoyo acercaron el lenguaje de programación C a las aulas del instituto San Felipe Neri de Cádiz.

Añadamos un cariñoso abrazo para nuestros familiares más cercanos, en especial a progenitores y abuelos, esos que en su día nos regalaron la llave que abría el portón de entrada al universo de la informática cuando tuvieron a bien obsequiarnos con lo que se convertirían en nuestros primeros ordenadores personales.

Por último, los autores quisiéramos agradecer a Paraninfo por la confianza depositada. Nuestra editora, Carmen Lara, ha mostrado una paciencia infinita que merece su justo elogio a través de estas líneas.

Sevilla, 4 de mayo de 2017

J. M. Maestre y J. Relinque

# Índice general

# Capítulo 1

# Introducción

Quizá no seamos plenamente conscientes de ello, en parte porque desde pequeños vivimos rodeados de todo tipo de artilugios electrónicos y ya forman parte de lo cotidiano, pero si uno se fija detenidamente en un ordenador cualquiera diría que funciona por arte de magia. ¿No es acaso mágico el hecho de que millones y millones de componentes electrónicos, tan pequeños que muchos son invisibles a la vista, conduzcan coordinadamente las señales eléctricas que dan vida a nuestros ordenadores?

Los ordenadores constituyen sin duda uno de los logros más importantes de la historia de la técnica y resulta fundamental conocer su funcionamiento por el impacto que tienen en nuestras vidas, tanto a nivel personal como profesional. Son incontables los problemas que pueden ser resueltos utilizando una computadora con la programación adecuada, algunos incluso sin necesidad de complejos conocimientos científicos.

Pongamos un ejemplo sencillo: se desea encontrar el valor mínimo que toma la función $f(x) = x^2$ para $x$ comprendido entre -5 y 5. Desde el punto de vista matemático, el problema es trivial si se sabe que el mínimo estará en la frontera o en un punto en el que la función tenga una derivada primera nula y una derivada segunda positiva. (Véase la figura 1.1 para comprobar que el mínimo se obtiene para $x = 0$.) Todo lo que necesita un informático para resolver el problema anterior es pedirle al ordenador que compruebe todos los posibles valores de $x$ existentes entre el -5 y 5 y que recuerde aquel que le ha proporcionado el valor de $f(x)$ más bajo. Indudablemente, la segunda solución es mucho menos elegante que la primera, pero es mucho más sencilla conceptualmente y no requiere siquiera conocer el concepto de derivada.

Del ejemplo anterior es posible extraer dos lecciones muy valiosas que os acompañarán en vuestra carrera como programadores. La primera es la siguiente: hasta el problema más complicado puede describirse mediante sencillas operaciones. Como vimos en el ejemplo anterior, examinar posibles valores de $x$ entre -5 y 5, calcular su $f(x)$ correspondiente y almacenar el valor mínimo hallado es un procedimiento muy sencillo. He aquí una excelente noticia: un ordenador es capaz de efectuar todas estas operaciones con extrema rapidez. Por esta razón la informática ha sido capaz de resolver problemas que hasta la fecha carecían de solución analítica, es decir, problemas

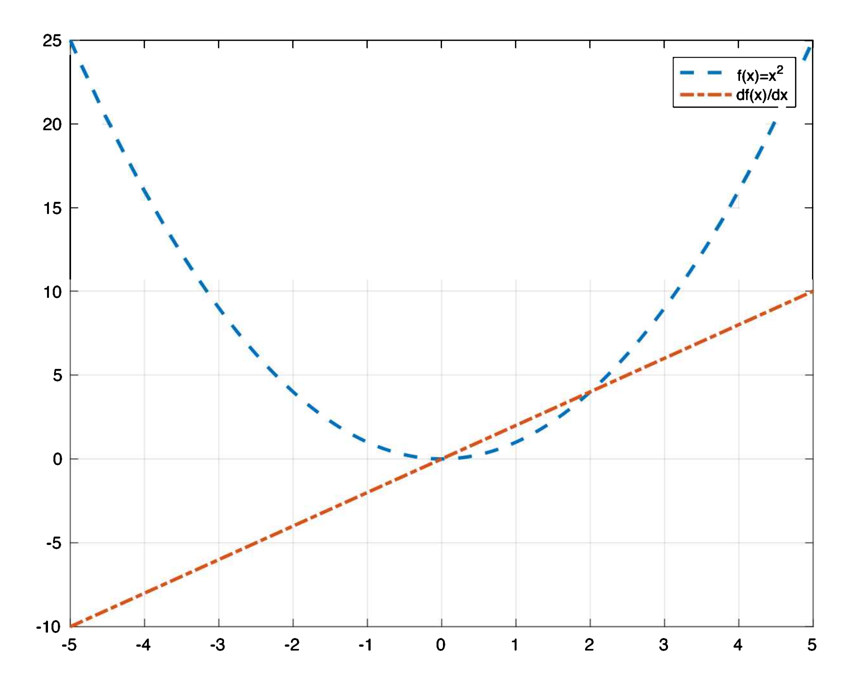



**Figura 1.1:** Representación gráfica de $f(x)$ y su derivada.

para los que los matemáticos no han encontrado una respuesta exacta y que solo han podido ser resueltos de forma numérica de manera similar a como hemos hecho con el mínimo de $f(x)$. Pero esto no quiere decir que cualquier problema pueda ser resuelto con la misma simplicidad gracias a un ordenador y tampoco conviene pasar por alto que cada problema tiene una serie de elementos que lo hacen único y que pueden ser explotados para simplificar la búsqueda de una solución. Uno piensa con lo que sabe, así que cuantos más conocimientos acumuléis mejores programadores seréis y mejores ideas tendréis.

Esto nos lleva a la segunda lección: existen muchas formas correctas diferentes de realizar un programa. Volviendo a nuestro ejemplo, una solución alternativa hubiera sido enseñar a la computadora a calcular las derivadas de $f(x)$ para encontrar así el mínimo. Otra solución válida para el problema consiste en comparar el valor de cada $f(x)$ con el de sus dos puntos vecinos. Si algún punto verifica que es menor que su vecino a la izquierda y a la derecha, tiene que ser un mínimo local. Si pensáramos un rato más seguro que se nos ocurrirían nuevas formas de resolver el problema. No obstante, no todas las soluciones son igualmente deseables; hay algunas más eficientes y rápidas que otras. Si algún día os contratan como programadores, será para que os enfrentéis a problemas difíciles, algunos de los cuales requerirán que el ordenador pase mucho tiempo calculando para encontrar una solución. Una programación ineficiente puede hacer que el ordenador tarde mucho más en hallar la solución o, peor aún, que se carezca de los recursos suficientes para calcularla.

## 1.1. Ordenadores

Es preciso que hablemos ahora de los elementos necesarios para que la informática nos ayude a progresar técnicamente. Lo único indispensable es un ordenador y un programador. Los programadores seréis vosotros, y este libro tiene como objeto ayudaros a crecer como tales. Por su parte, cualquier ordenador que utilicemos para programar contiene una serie de características básicas que deben conocerse. Dediquemos por tanto unas páginas a aprender algo más de los ordenadores que nos acompañarán en esta aventura.

### 1.1.1. Hardware y software, cuerpo y alma

A grandes rasgos un ordenador funciona gracias a dos tipos de elementos, el hardware y el software, vocablos ambos procedentes del inglés y que hacen referencia, literalmente, a lo *duro* y lo *blando* que hay dentro de una computadora. El hardware, lo *duro*, engloba a los componentes físicos del ordenador (pantalla, teclado...) mientras que el software representa algo tan intangible como necesario para su funcionamiento: la información que almacena y que determina cómo ha de comportarse (programas y datos asociados).

La forma más sencilla de distinguir entre ambos conceptos es aplicarlos a nosotros mismos. En una persona, el hardware sería el cuerpo, todo aquello que podamos tocar con los dedos (piel, órganos, etc.). El software sería lo que está almacenado en el cerebro, la información que tenemos grabada y que determina nuestra manera de pensar. Evidentemente, y aunque se trata de algo intangible, el pensamiento y los recuerdos son imprescindibles para *funcionar* como seres humanos.

En este libro nos centraremos en el pensamiento de las máquinas, la esencia de su alma, el software. Aprenderemos los fundamentos necesarios para poner a nuestro servicio los circuitos electrónicos de cualquier aparato susceptible de ser programado, ya sea un ordenador personal, una consola o un teléfono móvil. En otras palabras, seremos los magos que darán alma y, de algún modo, vida a las máquinas bajo nuestro control. Tened presente que sin un programa que dirija su funcionamiento estos dispositivos no serían otra cosa que caros y pesados pisapapeles.

## 1.2. El modelo de Von Neumann

Generalmente, la primera imagen que se viene a la cabeza de un ordenador consiste en una pantalla junto a un teclado y un ratón. Esta reducción es equivalente a pensar en una bicicleta en términos de su manillar o en un coche como un volante con parabrisas. Tal y como se decía en El Principito, “lo esencial es invisible a los ojos”, y esto es particularmente cierto en el caso de los ordenadores. Cualquier ordenador posee una serie de componentes que lo caracterizan como tal. Estos elementos fueron definidos por un matemático genial llamado John Von Neumann en 1945.

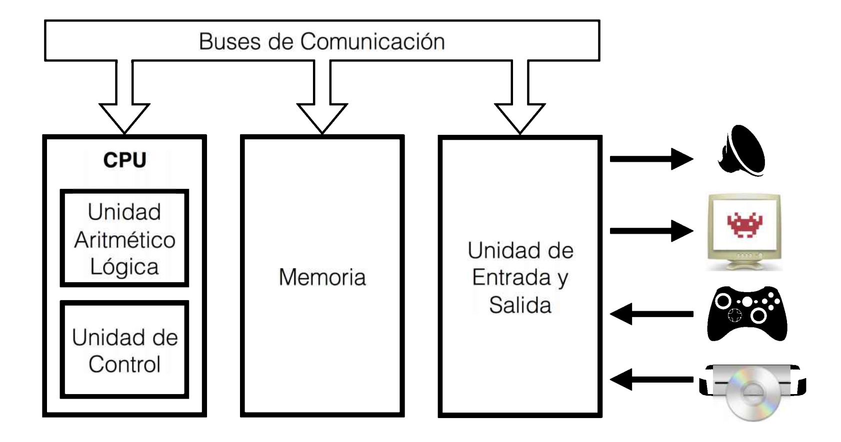



**Figura 1.2:** Esquema simplificado del modelo de Von Neumann.

Von Neumann fue un tipo con una vida de lo más interesante. No se puede esperar una vida muy convencional de una persona con un cociente intelectual de 180 (lo normal es rondar 100) e hipermnesia (una memoria casi ilimitada). Según sus allegados, uno podía leerle una frase al azar de un libro que hubiera leído y él mismo continuaría recitando su contenido prácticamente sin errores. Sus aportaciones fueron muy importantes en campos como matemáticas, física e informática. Son las contribuciones a este último campo las que más nos interesan aquí, en particular las relacionadas con la definición de la arquitectura básica de los ordenadores como los conocemos hoy día. Tal y como se muestra en la figura 1.2, dicha arquitectura comprende los siguientes elementos:

- **Unidad Central de Procesos**: es más conocida como CPU por el acrónimo del inglés *Central Process Unit*. Puede calificarse sin exagerar como el cerebro del ordenador ya que se encarga de coordinar a todos los componentes de la máquina y de realizar operaciones matemáticas.

  Este es uno de los elementos principales del ordenador, tal y como podéis comprobar en cualquier catálogo. Su característica principal es la frecuencia a la que realiza las operaciones, que se mide en hercios o hertz (Hz), lo que da una idea de su *velocidad*. De forma muy simplificada, un ordenador que funciona a 1 Hz puede realizar una operación por segundo. Los ordenadores actuales tienen velocidades de varios gigahercios, es decir, ¡pueden realizar miles de millones de operaciones por segundo!

  Dentro de la CPU pueden distinguirse a su vez dos componentes básicos:

  - Unidad de Control: es el director de orquesta de todos los elementos que componen el ordenador. Interpreta una partitura muy especial llamada programa, que contiene la secuencia de instrucciones que debe ejecutar la

máquina. Cada instrucción requiere la acción coordinada de varios elementos dentro de la computadora y es justamente la unidad de control la que se encarga de ello.

- Unidad Aritmético Lógica: es la calculadora del ordenador. Está compuesta por un conjunto de circuitos capaces de realizar operaciones de tipo aritmético (suma, multiplicación...) o lógico (mayor que, menor que, comparaciones...).

- **Memoria**: almacena la información que necesita el ordenador para su funcionamiento, la cual puede ser de dos tipos. En primer lugar están los datos, que es la información que el ordenador debe procesar[1]. En segundo lugar están los programas, esto es, la secuencia de instrucciones que el ordenador debe llevar a cabo.

- **Dispositivos de entrada y salida**: constituyen las vías de comunicación del ordenador con su entorno y sin duda se trata de los elementos más visibles del mismo. Los dispositivos de entrada permiten introducir información dentro del ordenador, por ejemplo, el teclado, el escáner o la *webcam*. Los dispositivos de salida permiten que el ordenador proporcione información a su entorno, por ejemplo a través de la pantalla, los altavoces o la impresora. Todos estos dispositivos se conectan a la computadora a través de las *interfaces* o adaptadores correspondientes.

- **Buses**: son cables que conectan los diferentes elementos de la máquina para poder transmitir información entre ellos. Para ahorrar cableado, todos los elementos se conectan a un mismo conjunto de cables o *bus*, que debe ser usado coordinadamente para evitar conflictos. Esto sería análogo a una residencia en la que todas las habitaciones están conectadas a través de un pasillo común. Cada vez que alguien quiere hablar con otra persona, no tiene más que asomarse al pasillo y gritar su mensaje, que será escuchado por todos y llegará con total seguridad a su destinatario. Para que este sistema de comunicación tan rudimentario funcione, es necesario que el resto calle mientras se transmite el mensaje. Lo mismo sucede dentro de un bus: los dispositivos conectados a él permanecen en silencio mientras alguno esté transmitiendo información.

  Hay varios tipos de buses, aunque los más importantes son:

  - Bus de datos: permiten llevar información de un punto a otro de la máquina. Por ejemplo si se quieren sumar dos números, hay que llevarlos de la memoria hasta la unidad aritmético lógica.
  - Bus de control: son utilizados por la unidad de control para indicar a los elementos de la máquina las tareas que deben realizar. Permiten, por ejemplo, indicar a la unidad aritmético lógica que debe llevar a cabo una operación de suma. Ello requiere que exista un cable que permita la transmisión física de esta orden desde la unidad de control hasta la unidad aritmético lógica.

[1] La palabra *informática* está asociada etimológicamente a "información" y "automática", en relación al tratamiento o procesamiento automático que se realiza de una cierta información.

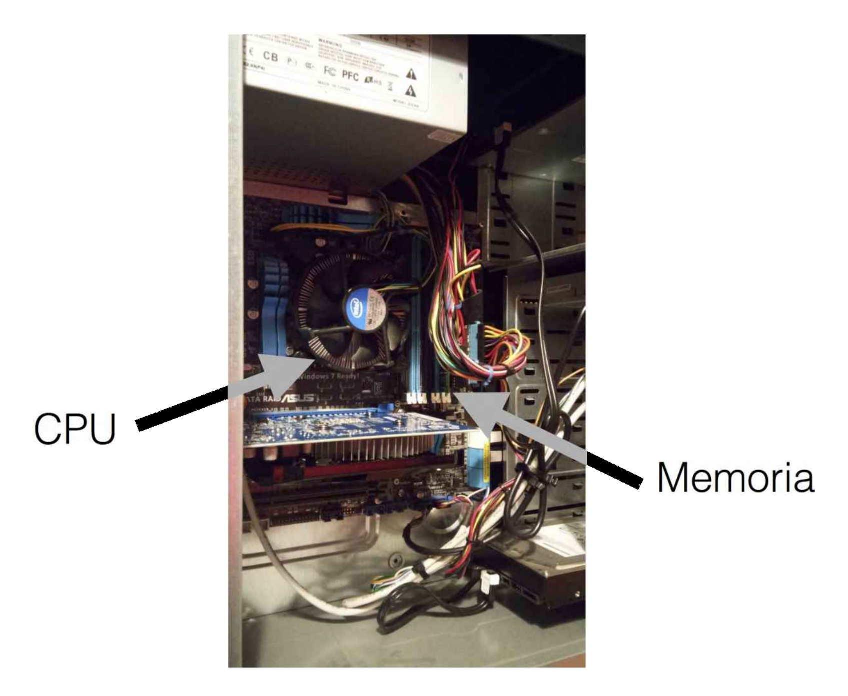



**Figura 1.3:** Interior de la torre de un ordenador de sobremesa.

Finalmente, en la figura 1.3 se muestra el interior de un ordenador de sobremesa. La CPU de este equipo es refrigerada por un ventilador en el que puede leerse el nombre del fabricante *Intel*. A su derecha se aprecian unas ranuras alargadas en las que se insertan los módulos de memoria principal. En este caso, el más cercano a la CPU está vacío, aunque el siguiente, tapado parcialmente por una maraña de cables, tiene su módulo de memoria insertado. Ambos componentes se encuentran conectados a la *placa base* del ordenador, la cual les proporciona soporte y conectividad. Bajo la CPU se observa una tarjeta conectada perpendicularmente a la placa base. Se trata de la tarjeta gráfica. Dentro del esquema de Von Neumann, puede verse como una interfaz de salida del ordenador, ya que permite conectar una pantalla al mismo. En la parte superior de la fotografía se aprecia la fuente de alimentación, que alimenta eléctricamente a los dispositivos de la computadora y que es responsable parcialmente de la maraña de cables anteriormente indicada. El resto de cables son buses para el intercambio de información entre los dispositivos de la máquina.

## 1.3. Codificación de la información

Los ordenadores funcionan gracias a la electricidad que da vida a los millones de componentes electrónicos que albergan en su interior. Que semejante cantidad de componentes funcione correcta y coordinadamente es un reto que todavía hoy mantie-

ne ocupados a los ingenieros, pues constatemente producen innovaciones destinadas a mejorar la velocidad de los computadores. El problema se complica aún más cuando se tiene en cuenta que todos los circuitos producidos por la industria deben comportarse de la misma manera. Por ejemplo, Intel debe garantizar que todas las unidades producidas de cualquier modelo de CPU se comportan de la misma manera, y además debe garantizarlo para un amplio rango de condiciones ambientales. Merece la pena destacar la importancia de este último matiz: el comportamiento de la electricidad dentro de un circuito se ve afectado por condiciones ambientales como la temperatura. Sin embargo, todos esperamos que nuestros ordenadores funcionen igual de bien en el caluroso verano de Sevilla y en el frío invierno de Soria. Lo contrario simplemente no sería aceptable.

Por estos y otros motivos, la tecnología informática siguió un camino que premiaba la robustez y la simplicidad en el funcionamiento frente a otros factores. En términos de electricidad, lo simple es pensar en términos de todo o nada: un interruptor está encendido o apagado, un cable puede conducir o no conducir corriente, el nivel de voltaje en un condensador puede ser alto o bajo... Esta dicotomía ofrece una robustez natural que soluciona muchos de los problemas mencionados anteriormente. Poco importa que la temperatura varíe la cantidad de corriente que circula en un cable si todo lo que interesa saber es si el cable conduce corriente o no. En el mundo de los ordenadores todo es *binario* y, por tanto, todo queda confinado dentro de dos valores posibles. Por convención, uno de los valores se representa con 1 y el otro con 0, lo que permite abstraerse de la implementación física del circuito para pensar únicamente en términos de unos y ceros. Al tipo de circuitos que están diseñados para trabajar de forma binaria se los conoce como circuitos lógicos o digitales.

Si toda la información que procesa el ordenador es reducida en última instancia a unos y ceros, ¿cómo es posible que una película o una canción puedan ser almacenadas dentro del mismo? La respuesta a esta pregunta se esconde detrás de algo tan natural como contar con los dedos. A fin de cuentas, hasta el hecho más sencillo esconde implicaciones profundas si se reflexiona lo suficiente. Por ejemplo, pensemos en los dedos de una mano. ¿Hasta qué número se puede contar? Una mano, cinco dedos. Parece obvio que se puede contar hasta el número cinco. En realidad, bajo esta respuesta se esconde la convención de que para contar con las manos se suman simplemente el número de dedos levantados. De esta manera se desecha información que podría ser aprovechada para representar otros números. Por ejemplo, si quiere representar el 2, da lo mismo levantar los dedos índice y anular que el meñique y el pulgar. Ambas combinaciones se identifican con el 2 según la convención establecida para contar con los dedos. Sin embargo, con una única mano pueden realizarse hasta 32 combinaciones diferentes con los dedos. Es fácil llegar a esta conclusión: cada uno de los cinco dedos puede estar extendido o recogido, es decir, en una de dos posiciones posibles. Por tanto, cada dedo multiplica por 2 el número de posibilidades, lo que da un total de $2^5$ combinaciones, o sea, 32.

Realicemos una nueva convención para contar a partir de este momento. Cuando todos los dedos de la mano estén recogidos denotaremos el número cero. Levantar el pulgar equivale a sumar 1 ($2^0$), el índice suma 2 ($2^1$), el corazón 4 ($2^2$), el anular 8 ($2^3$)

y el meñique 16 ($2^4$). Realizad pruebas levantando unos dedos y bajando otros. Comprobaréis que podéis generar todos los números del 0 al 31, esto es, 32 combinaciones diferentes. Si en lugar de aplicar este sistema a una mano lo hiciéramos con las dos, podríamos generar ¡1024 combinaciones diferentes! Es decir, podríamos contar *todos* los números del 0 al 1023 con los dedos siguiendo este procedimiento. (Los dedos de la segunda mano tendrían que valer respectivamente $2^5$, $2^6$, $2^7$, $2^8$ y $2^9$ para que el truco funcione).

En informática, uno no cuenta con dedos sino con *bits*, abreviación del inglés *binary digit*, esto es, dígito binario. Un bit es un número que puede valer 0 o 1 y su valor representa la mínima cantidad de información que puede ser almacenada. Con un solo bit puede representarse una de dos posibilidades. Con cinco bits pueden representarse $2^5$ combinaciones posibles, ya que cada bit, al igual que cada dedo de la mano, puede tomar uno de dos valores. En general, con $n$ bits pueden formarse $2^n$ combinaciones diferentes.

Es muy importante ser conscientes de que estas combinaciones de bits pueden tener diferentes significados. Todo depende del convenio que se siga. De la misma forma que uno puede poner dos dedos en forma de V para representar el 2 o la señal de victoria, en informática pueden usarse los bits para codificar números u otras cosas. Por ejemplo, las 256 combinaciones que pueden formarse con 8 bits pueden servir para contar del 0 al 255, para representar todos los caracteres del teclado –incluyendo letras, números y símbolos– o para indicar la intensidad del gris que tiene un píxel en una imagen en blanco y negro. Todo depende de la codificación empleada, es decir, del conjunto de reglas utilizado para establecer la correspondencia entre los bits y la información que se quiere representar. Esta versatilidad es la que permite almacenar como unos y ceros cosas tan dispares como música, películas, textos o videojuegos.

Finalmente, y sin entrar en más detalles, conviene tener presente que toda la circuitería del ordenador está optimizada para trabajar de esta manera. Por ejemplo, el ordenador suma o multiplica gracias a su unidad aritmético lógica, que está diseñada para realizar estas operaciones con conjuntos de bits. Los datos que se almacenan en la memoria o en el disco duro, también son almacenados en forma de bits. Como ya dijimos, dentro de la máquina el sistema binario lo define todo.

## 1.4. Los idiomas de las máquinas

Como se verá, programar consiste en crear recetas con los pasos que el ordenador debe seguir para resolver un determinado problema. Nuestros programas se almacenarán dentro de la memoria y serán interpretados por la unidad de control. Por ello, con cada programa que escribimos, de alguna manera somos nosotros los que estamos gobernando la máquina, lo que hace que la programación se convierta en una especie de magia que permite que tomemos el control de un ordenador, el cual estará a nuestro servicio y hará lo que le pidamos siempre que sepamos utilizarlo convenientemente.

Pero, ¿en qué lenguaje se habla con una máquina?

Hace muchos años, cuando los primeros ordenadores fueron desarrollados, los programadores tenían que hablar con las máquinas en su propio lenguaje de unos y ceros. Por desgracia para ellos, la mente humana no está especialmente preparada para trabajar así, de modo que la programación de aquellas máquinas era un auténtico suplicio. Para hacerse una idea más concreta de la dificultad, veamos una instrucción real de un procesador de Intel:

```
1011 0000 0110 0001
```

Esta ristra de unos y ceros, ejecutada en una CPU de la familia x86 de Intel, carga información dentro la CPU. Contrariamente a lo que pueda parecer, los programadores no son masoquistas, de manera que se pusieron manos a la obra para simplificar la titánica tarea de programar en estas condiciones. El primer paso que se dio en este sentido fue el de desarrollar software que tradujera a lenguaje máquina el código que escribían. En un primer momento, se optó por sustituir las ristras de ceros y unos por palabras que tuvieran una correspondencia directa. Así se llegó al lenguaje ensamblador, donde la ristra anterior puede expresarse como:

```
MOV AL, 061h
```

El ejemplo muestra con claridad que el lenguaje ensamblador no es muy intuitivo. No obstante, pese a lo críptico que pueda llegar a ser, es más inteligible que su equivalente binario, lo que agiliza tanto la escritura como la depuración de los programas. Resulta mucho más sencillo para el ser humano trabajar en lenguaje ensamblador que en código máquina, todo ello con la enorme ventaja de que al trabajar al mismo nivel que el propio procesador es posible disfrutar de un control completo de los recursos de la máquina. No hay nada tan eficiente en términos de tiempo de ejecución como un programa realizado en ensamblador. Esta fortaleza se convierte también en su principal debilidad: hay que indicar a la máquina *todo* lo que tiene que hacer. Ello hace que hasta el programa más sencillo requiera varias decenas de líneas de código en ensamblador. Otra dificultad adicional es que es muy dependiente del hardware de la máquina, que puede variar sustancialmente entre diferentes modelos de ordenador.

En general, y siendo un poco más estrictos, se llama lenguajes de bajo nivel a aquellos lenguajes de programación que se encuentran muy cercanos a lo que la máquina realmente entiende y los programas que realizan la labor de conversión de lenguaje de bajo nivel a lenguaje máquina se denominan ensambladores. Por extensión, un lenguaje de bajo nivel también puede ser llamado lenguaje ensamblador.

En este punto merece la pena dedicar unos momentos a apreciar en toda su magnitud la abstracción que supone crear un programa que recibe como entrada un cierto texto en ensamblador y genera a partir de su procesamiento otro programa de ordenador. ¡Un programa que crea programas! Se trata de un logro verdaderamente importante. Tanto, que ha sido repetido varias veces más a lo largo de la historia de la informática y ha permitido que hoy disfrutemos de lenguajes de programación mucho más cercanos al lenguaje humano. Al fin y al cabo, si se puede convertir el ensamblador en binario, ¿por qué no crear otros lenguajes más cercanos al ser humano para traducirlos a ensamblador y de ahí a código máquina? Sin entrar mucho en los detalles, es posible construir un programa que traduzca del nuevo lenguaje de progra-

mación a otro de menor nivel tantas veces como sea preciso hasta llegar al lenguaje máquina.

En contraposición a los lenguajes de bajo nivel, se conoce como lenguajes de alto nivel a aquellos que permiten que los programadores expresen sus órdenes de manera mucho más cercana al lenguaje humano. Por ejemplo, el lenguaje de programación BASIC, diseñado en 1964, permite utilizar instrucciones como esta:

```
PRINT "Hola"
```

En caso de que no lo hayáis adivinado aún, esta instrucción imprime `Hola` por pantalla. Como puede verse, BASIC es mucho más sencillo de entender (y dominar) que cualquier lenguaje ensamblador. Entonces, ¿por qué no programar directamente en lenguajes de alto nivel si ya disponemos de programas que traducen a lenguaje máquina? Sin duda sería mucho más sencillo aprender a programar de esa manera.

Lamentablemente, programar con lenguajes de alto nivel tiene un alto coste en términos de eficiencia. Para cuando nuestro código original ha sido traducido a lenguaje máquina se ha añadido mucho código extra que nunca hubiéramos introducido de haber realizado el programa directamente en lenguaje ensamblador. Por ejemplo, imaginemos que para hablar con un ruso tuviéramos que contratar a un traductor de español a inglés, otro de inglés a alemán, y por último a uno de alemán a ruso. Para que lo que digamos no llegue distorsionado al receptor es necesario incluir información extra en cada paso de traducción, de forma que se recojan todos los matices de lo expresado inicialmente. Algo similar sucede cuando se utilizan lenguajes de alto nivel: a la máquina llega más código del indispensable, tanto más cuanto mayor sea la cercanía del lenguaje de programación al humano. Pese a todo, esta pérdida de eficiencia es admisible en muchas situaciones. Por ejemplo, si se necesita un programa que calcule el área de un círculo a partir de su radio, da igual que el cálculo se haga en 1 o en 0.1 segundos. Por el contrario, si lo que se pretende realizar es un programa que gobierne la trayectoria que sigue una nave espacial en su desplazamiento, la pérdida de eficiencia puede llegar a ser inaceptable. Por ello, existen multitud de lenguajes de programación, de modo que el programador puede escoger el que más le interese en términos de eficiencia y facilidad de uso según la aplicación que tenga en mente.

Por último, deberíamos reflexionar sobre lo que supone aprender un *lenguaje de programación*. La propia palabra "lenguaje" indica con claridad lo que os espera en esta aventura: aprenderéis una nueva lengua para comunicaros con las máquinas, lo que requerirá memorizar un vocabulario básico y aprender una sintaxis que os permita expresaros de forma coherente. Como todo lenguaje, la manera más sencilla de aprenderlo es practicarlo, ¿o es que conocéis a alguien que hable inglés con fluidez sin haberlo puesto en práctica muchas veces? Afortunadamente, el esfuerzo de aprender a programar es considerablemente menor que el de aprender a hablar inglés. Con unas decenas de palabras y algunas sencillas reglas podréis utilizar la magia de la que hablábamos en la introducción. Sin programadores, los ordenadores serían poco más que una triste caja con la pantalla en negro, así que si queréis aprender a darles vida, si queréis saber cómo crear "hechizos" para gobernar las máquinas, no os perdáis la siguiente sección porque empezamos a aprender C.

## 1.5. El lenguaje de programación C

En este libro aprendemos a programar en el lenguaje de programación C. Se trata de un lenguaje que fue creado en 1972 por Dennis M. Ritchie en los laboratorios Bell. A pesar de su antigüedad se trata de un lenguaje muy popular y sigue siendo muy utilizado. Podría decirse que C es a los lenguajes de programación lo que los Beatles son a la música.

La importancia de C es tal que la sintaxis de muchos lenguajes actuales como PHP, Java o C# es muy similar. Más aún, la didáctica familia de microcontroladores Arduino se programa en C. Por lo tanto, aprender C simplifica el aprendizaje de otros lenguajes y la programación de todo tipo de dispositivos en el futuro.

En cuanto a su complejidad, C es un lenguaje de programación de nivel medio, esto es, no es tan cercano al programador como BASIC pero está lo suficientemente alejado de la máquina como para que su aprendizaje no suponga un problema. Asimismo, se trata de un lenguaje extremadamente versátil, que permite que el programador pueda acceder a los recursos de la máquina a bajo nivel en caso de que sea necesario. Gracias a ello, el código en C es convertido en lenguaje máquina con mucha eficiencia.

### 1.5.1. La memoria vista desde C

De todos los elementos del modelo de Von Neumann, la memoria es el más importante debido a que es en ella donde se depositan en última instancia nuestros programas y los datos que tengan asociados. Los programas se componen de una serie de instrucciones que indican al procesador o CPU las tareas que tiene que llevar a cabo. La mayoría de estas operaciones son muy sencillas: sumar dos números, leer la información contenida en una determinada posición de memoria... Los datos, por lo general, son información necesaria para que esas instrucciones puedan ser llevadas a cabo. Por ejemplo, resulta razonable asumir que si se va a pedir al procesador que sume dos números es porque están almacenados en algún lugar, y que además se dispone de alguna zona de memoria donde almacenar el resultado de la operación.

Veamos ahora una metáfora para explicar el funcionamiento de la memoria y su relación con el procesador. Los protagonistas de nuestra metáfora serán dos: un lugar y un personaje:

- La memoria: imaginaos que ante vosotros se extiende una hilera de armarios larguísima, tan larga que parece una calle, la calle memoria. Esta calle particular está repleta de armarios del mismo tamaño. Cada uno de estos armarios tiene un número único asignado, su dirección en la calle memoria, y contiene dentro un papel con información.
- La CPU: el procesador es un tipo muy obediente que se dedica a hacer todo aquello que encuentra escrito dentro de los armarios de la memoria. Al encender la computadora, el procesador empieza siempre abriendo la primera puerta, lee

lo que pone en su interior y lo ejecuta. Mientras no le digan lo contrario, el procesador abrirá un armario tras otro de forma consecutiva, leerá el contenido del papel en su interior y ejecutará la instrucción correspondiente.

Examinemos el ejemplo de la suma que comentábamos antes a la luz de nuestra metáfora. El procesador está en plena jornada laboral, lleva un rato ya trabajando y digamos que le toca leer a continuación la información detrás de la puerta número 224. Como siempre, se dirige hasta ella y la abre. En su interior encuentra un papel donde puede leerse "suma los contenidos de los armarios 2542 y 343". El procesador acude obedientemente a estas dos celdas y encuentra dentro dos papeles en los que lee, respectivamente, "12" y "38". Esta me la sé, piensa, suman 50. Para que no se le olvide el resultado se lo apunta en una chuleta que siempre lleva encima y que recibe el nombre de registro. Como ya ha terminado de ejecutar lo que decía el papel de la posición de memoria 224, se va a la 225 para continuar con su labor. La abre y encuentra escrito "guarda el resultado anterior en la celda 3500". A continuación, el procesador corre hasta la celda 3500, tira el papel que había dentro sin mirarlo siquiera y deja allí un papel con lo que tenía escrito en la chuleta, en este caso "50".

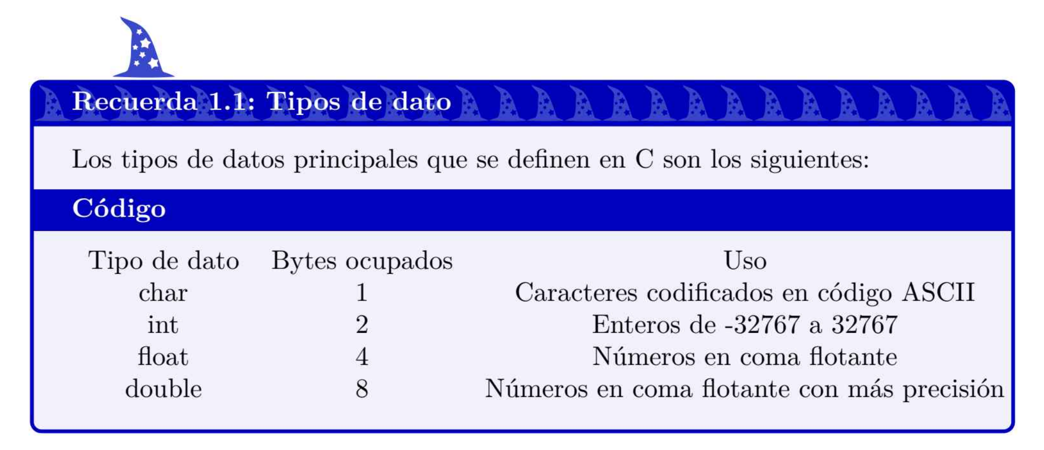
### 1.5.2. El tamaño del papel es limitado: tipos de dato

El papel dentro de cada posición de memoria tiene un tamaño limitado. Uno no puede escribir ahí lo que quiera. Hay unos límites de tamaño que deben ser conocidos y respetados. Además, como ya dijimos, existen diferentes reglas de codificación para representar información con los unos y los ceros. Todo esto nos lleva a presentar a un importante concepto en C, el de *tipo de dato*.

Empecemos con lo primero, la cantidad de información que se almacena en una posición de memoria. Por ejemplo, si en el papel de nuestra metáfora solo cabe 1 byte (8 bits), podrán representarse 256 cosas diferentes como ya vimos anteriormente. La interpretación de la información que haya ahí ya es cosa nuestra; lo único claro es que hay 256 posibilidades distintas que van del 00000000 al 11111111. La pregunta es si con 256 posibilidades somos capaces de codificar lo que queremos, algo que, por supuesto, depende de lo que queramos. Por ejemplo, el código ASCII asigna un carácter a cada una de las 256 posibilidades. Dado que hay muchas más posibilidades que letras en el alfabeto se codifican también otros caracteres como los números (del '0' al '9'), signos de operaciones matemáticas ('+', '-', '*' y '/') e incluso algunos emoticonos hasta completar 256 caracteres diferentes. Es decir, al código ASCII le basta con 8 bits.

¿Y si lo que queremos es contar los números entre el 0 y el 65535? Entonces, lamentablemente, las 256 posibilidades se quedan cortas. La solución a este problema es sencilla: basta utilizar más de una posición de memoria para almacenar mi información. Si en uno de nuestros armarios caben 8 bits, dos almacenarán 16 bits. Esto se traduce en 65536 posibilidades diferentes que van desde el 0000000000000000 al 1111111111111111. Bingo. Con dos posiciones hay posibilidades suficientes para asignar a cada una el significado de los números desde el 0 al 65535.

Por tanto, las reglas de codificación que se utilicen deben contemplar dos aspectos: el tamaño empleado (medido en bits) y la interpretación que se hace de los unos y ceros correspondientes. Esto es precisamente lo que el concepto de tipo de dato representa.



**Recuerda 1.1: Tipos de dato**

Los tipos de datos principales que se definen en C son los siguientes:

**Código**

| Tipo de dato | Bytes ocupados | Uso |
|---|---|---|
| char | 1 | Caracteres codificados en código ASCII |
| int | 2 | Enteros de -32767 a 32767 |
| float | 4 | Números en coma flotante |
| double | 8 | Números en coma flotante con más precisión |

Evidentemente, estos tipos de dato tienen limitaciones derivadas del tamaño que ocupan en memoria. Por ejemplo, siempre encontraremos un número lo suficientemente grande o con la suficiente cantidad de decimales como para no poder ser almacenado con total exactitud en memoria. Este tipo de errores numéricos son habituales en programación. Para paliar este problema se introducen algunos modificadores como `long`, que antepuesto al tipo de dato `int` habilita algunos bytes extra con el objetivo de almacenar números más grandes. Existen otros modificadores que cambian las propiedades de los tipos de dato en función de las necesidades del programador, aunque su utilización no se verá en este libro, ya que suele quedar restringida a problemas muy específicos.

### 1.5.3. Variables

En nuestra metáfora sobre el funcionamiento de la memoria vimos cuán obedientemente iba el procesador al armario con la dirección 3500 cuando se lo solicitábamos. Esto puede resultar muy intuitivo para el procesador, que está diseñado para trabajar con números, pero no para la mente humana. He aquí la razón de ser de las variables: resulta muy conveniente para el programador identificar con nombres ciertas posiciones de memoria. Por ejemplo, supongamos que bautizamos al armario de la dirección 3500 y le ponemos una plaquita con un nombre escrito en ella. Por comodidad supondremos que el nombre es el de una persona, como Juan, pero la etiqueta puede ser cualquier cadena alfanumérica que respete unas normas básicas que veremos a continuación. Está claro que para cualquiera de nosotros es más fácil pedirle al procesador que compruebe lo que hay en el armario con el rótulo Juan que pedirle que vaya a la dirección 3500. Ambas cosas implican lo mismo, pero la primera es más fácil de recordar.



**Recuerda 1.2: Reglas para nombrar variables**

Los nombres de las variables han de respetar las siguientes normas:

**Código**

- Deben empezar con una letra. A partir de esa posición el resto de caracteres han de ser alfanuméricos (letras del alfabeto inglés o números), aunque también se admite el carácter guión bajo '_'.
- En el lenguaje C las letras mayúsculas y minúsculas cuentan como caracteres diferentes, es decir, los nombres de variables Juan, juan, juAn y JUAN son todos diferentes. Por esta característica se dice que C es *case sensitive* (sensible a las mayúsculas). Ojo con esto.
- Ninguno de los nombres podrá coincidir con el de las siguientes palabras, cuyo uso queda reservado por tener significado propio en C.

```
auto    break   case    char    const   continue
default do      double  else    enum    extern  float
for     goto    if      int     long    register  return
short   signed  sizeof  static  struct  typedef
union   unsigned  void  volatile  while
```

Todo lo que se requiere en C para definir una variable es un tipo de dato y un nombre. No se necesita nada más. Así pues ya estáis en condiciones de aprender vuestra primera regla sintáctica dentro de este lenguaje.



**Recuerda 1.3: Declaración de variables en C**

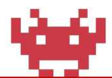
Para declarar una variable basta escribir una línea de código con la estructura que se indica a continuación.

**Código**

```
tipo_de_dato   nombre_variable;
```

Es decir, utilizar uno de los armarios de la calle memoria requiere únicamente indicar el tipo de información que se va a almacenar en su interior y proporcionarle un nombre, un rótulo para que quede bien claro el nombre con el que nos referiremos a su contenido dentro del programa. La instrucción concluye con el carácter punto y coma (';').

Veamos un ejemplo para entender mejor la forma en la que se declaran variables en C.



**Ejemplo 1.1: Declaración de variables**

Este ejemplo muestra cómo podrían declararse diferentes variables en C para albergar datos de tipo entero, flotante y carácter.

**Código**

```
float radio;
int aux=3;
int ancho, largo=1;
char letra='a';
```

Cualquiera de las instrucciones del ejemplo 1.1 reservará un trozo de memoria que esté libre y la etiquetará con el identificador que hayamos suministrado. En concreto, se reserva espacio en la memoria para almacenar:

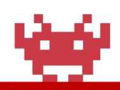
- Una variable llamada `radio`, de tipo flotante.
- Una variable llamada `aux`, de tipo entero y con valor inicial 3.
- Dos variables enteras llamadas `ancho` y `largo`, esta última con valor inicial 1.
- Una variable llamada `letra` con valor inicial 'a'. (El uso de comillas simples es obligatorio para enfatizar que se trata de un carácter).

La zona de memoria reservada no será siempre la misma entre ejecuciones sucesivas del programa. La razón principal es que un programa puede recibir una zona de memoria diferente cada vez que es ejecutado. El encargado de asignar a cada programa la zona de memoria en la que será ejecutado es el sistema operativo. ¿Y quién es el sistema operativo para tener semejante privilegio? Pues es el programa con mayúsculas; se está ejecutando siempre desde que se enciende el PC y hace las veces de director de orquesta a nivel de software. Gestiona los recursos del computador y aloja a los programas en zonas libres de memoria para que se ejecuten. Como las zonas libres dependen de lo que se esté ejecutando en ese momento en el ordenador, es imposible conocer a priori las direcciones que tendrán las variables. No obstante, si queremos hacer referencia a la dirección de memoria en la que se almacena una cierta variable en C basta anteponerle el operador ampersand `&`. Por ejemplo, `&largo` representa dentro del código la dirección en la que se almacena la variable `largo`.

El ejemplo 1.1 muestra también que es posible declarar varias variables a la vez y asignar un valor inicial cuando se estime oportuno. Aquellas variables no inicializadas están definidas en memoria aunque su valor es desconocido. En términos de nuestra metáfora, esto significa que aunque el armario tiene el rótulo con el nombre puesto no sabemos qué es lo que pone en el papel que hay en su interior. Por tanto, nunca debe utilizarse una variable a la que no se le haya dado primero un valor conocido,

ya que de lo contrario estaremos introduciendo un grado de incertidumbre indeseable en nuestros programas.

La asignación y la operación con variables en C también es extremadamente sencilla. Basta igualar el nombre de la variable con un valor o una expresión. Por supuesto, la variable tiene que haber sido declarada o de lo contrario no podrá almacenar un valor. Veamos un ejemplo.



**Ejemplo 1.2: Asignación y operaciones con variables**

Este ejemplo muestra algunas operaciones sencillas en C con variables.

**Código**

```
int area, ancho, largo;
ancho =2;
largo = 4*2/4+1;
area=ancho*largo;
```

En este ejemplo se declaran las variables **area**, **ancho** y **largo** y se les asignan valores diferentes mediante el operador **=**. En C se evalúa primero lo que está a la derecha de la igualdad y el resultado que se haya obtenido se asigna a la variable indicada en la parte izquierda de la misma. En este ejemplo **area** vale 6, ya que **ancho** vale 2, y **largo** vale 3.

## 1.6. A programar se aprende jugando

Hasta aquí el único capítulo teórico del libro. Aunque no es imprescindible, sí es recomendable conocer todo lo que hemos explicado. Estos fundamentos acerca de la manera en la que funcionan los ordenadores os ayudarán a entender mejor todo lo que sigue a partir de ahora. Empezáis una gran aventura y os invitamos a que practiquéis mucho para convertiros en magos de la programación. Aprended de los ejemplos y los juegos que os mostramos pero poned desde ya el motor de la imaginación en marcha. Diseñad vuestros propios juegos y convertidlos en realidad. A programar se aprende programando, sí, pero también jugando.

# Capítulo 2

# Interacción básica con el PC

En el presente capítulo vamos a intentar poner en marcha una serie de procesos cuyo objetivo central sea el poder interactuar con nuestro ordenador. En el mundo real, la interacción con las personas se realiza a través de mensajes; estos mensajes pueden transmitirse de múltiples maneras, aunque, en realidad, lo que cambia es la vía de comunicación. Podemos decirle algo a otro ser humano a través de una palabra, un gesto, un mensaje escrito, o incluso de uno o varios iconos, algo que se ha puesto de moda en los últimos tiempos gracias a las aplicaciones de mensajería instantánea que ejecutan los teléfonos inteligentes.

En el caso de simular una comunicación con la máquina, el lenguaje de programación C provee de una serie de comandos con los que se establecerá una especie de medio de comunicación con el ordenador. El objetivo final puede ser todo lo variado que queramos. Por ejemplo, la computadora puede realizar un anuncio que nos interese a través de un mensaje por pantalla. O quizás sea posible que introduzcamos nuestro nombre mediante el teclado. De esa manera, el ordenador podrá dirigirse hacia nosotros de forma educada.

Pero lo más relevante del capítulo en el que nos encontramos radica en las decisiones basadas en condiciones. La computación se encarga de abstraer y modelar los problemas que nos encontramos en el día a día, de manera que un ordenador pueda llegar a entenderlos y resolverlos. A la hora de diseñar un algoritmo que implemente un proceso cualquiera, la toma de decisiones será un elemento crucial, ya que el ordenador será capaz de realizar una u otra acción en función de los datos que maneje.

## 2.1. El ordenador sabe hablar: printf, puts y putc

La función de salida estándar más utilizada en lenguaje C es `printf()`. El objetivo de dicha función –`printf` es una abreviatura de print format: imprimir con formato– es presentar a través de la salida estándar –normalmente, la pantalla– un mensaje determinado, de modo que el usuario pueda leerlo con claridad. La potencia de esta

función radica en la posibilidad de indicar el tipo de formato que tiene nuestro mensaje. Para ello, **printf()** presenta la siguiente sintaxis, en la que se puede apreciar que recibe dos tipos de argumentos.



**Recuerda 2.1: Sintaxis printf()**

```
printf ("Cadena de control", lista de datos separados por comas);
```

Detallamos a continuación los argumentos que recibe la función:

1. El primer argumento es la cadena de control, que contendrá dos tipos de elementos: una cadena de texto para presentar tal cual por pantalla y uno o varios especificadores de formato, los cuales determinarán el formato en el que aparecerán los valores especificados en el segundo argumento (debe haber tantos especificadores de formato como datos haya).

   Los especificadores de formato más usuales son los siguientes:

   - `%c` formato de carácter.
   - `%s` formato de cadena de caracteres.
   - `%d` formato de entero decimal.
   - `%f` formato en coma flotante.

2. El segundo argumento es opcional e incluye una sucesión de datos separados por comas. Se deberán incluir tantos datos como especificadores de formato se incluyan en el primer argumento. Cada uno de estos datos deberá coincidir en formato con el tipo de dato indicado por el especificador correspondiente.

   Hay que tener en cuenta que **printf()** se engloba dentro de la librería de entrada y salida estándar por excelencia de C: stdio.h. Se trata de un archivo de cabecera. Así, cada vez que vayamos a escribir un programa que utilice **printf()**, será totalmente indispensable incluir dicha librería al comienzo del código mediante la directiva **#include**.



**Ejemplo 2.1: Bienvenido al sistema**

En primer lugar, exhibiremos buenos modales y realizaremos un programa en C que muestre al usuario en pantalla el siguiente mensaje: Bienvenido al sistema. Al finalizar el texto se añadirá un carácter de escape, que agregará un salto de línea. En este caso no se utiliza el segundo argumento.

**Código**

```
#include <stdio.h>
main()
{
    printf("Bienvenido al sistema \n");
}
```



**Ejemplo 2.2: Imprimiendo números**

En pantalla aparece El número 18. El segundo argumento es un entero decimal, tal y como se especifica en el primer argumento mediante el indicador %d

**Código**

```
#include <stdio.h>
main()
{
    printf("El numero %d", 18);
}
```

Aparte de `printf()`, el lenguaje C permite usar dos funciones más de escritura por salida estándar, las cuales resultan más directas y específicas para un formato concreto. En primer lugar, `putc()` se encarga de volcar por pantalla un único carácter. Su sintaxis es muy sencilla, requiriendo dos parámetros:

**Recuerda 2.2: Sintaxis putc()**

```
putc(caracter, salida);
```

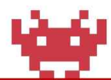
El primer parámetro es el carácter a imprimir, mientras que el segundo es el flujo de salida al que quiere volcarse dicho carácter. Lo habitual será que este último parámetro sea `stdout`, o sea, la salida estándar.

En una línea parecida se encuentra `puts()`, que volcará por pantalla una cadena de caracteres. De nuevo, la sintaxis no requiere de especificadores de formato ni nada parecido, puesto que está totalmente orientada a los textos:



**Recuerda 2.3: Sintaxis puts()**

```
puts(cadena);
```



**Ejemplo 2.3: Misterioso saludo**

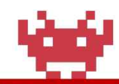
Para practicar las funciones que acabamos de aprender imprimiremos por pantalla un carácter y un saludo para dar la bienvenida al usuario misterioso X.

**Código**

```
#include <stdio.h>
main()
{
    char c = 'X';
    puts("Bienvenido al sistema,");
    putc(c, stdout);
}
```

## 2.2. El origen de “Hola, mundo”

Para todo programador que desea adentrarse en un nuevo lenguaje existe una primera meta que debe alcanzar: escribir un programa que haga que el ordenador muestre por pantalla el mensaje `Hello, world`. Se trata de una tradición cuyo origen se encuentra en un libro escrito por Brian Kernighan en 1973. La obra en cuestión se llamaba *A Tutorial Introduction to the Programming Language B* y abordaba una pequeña introducción al lenguaje B, que es considerado un predecesor directo de C. Kernighan, que escribiría cinco años después The C Programming Language junto con el creador de C, Dennis Ritchie, inventó un pequeño ejercicio en el que, mediante unas cuantas líneas, la computadora saludaba al mundo desde el monitor.



**Ejemplo 2.4: Hola mundo**

Para no faltar a la tradición, escribiremos un programa completo que muestre por pantalla el conocido mensaje Hola Mundo mediante el uso de printf() y un par de especificadores de formato de cadena de texto.

**Código**

```
#include <stdio.h>
main()
{
    printf("%s %s", "Hola", "Mundo");
}
```

Para saber más: `http://blog.hackerrank.com/the-history-of-hello-world/`.

## 2.3. El ordenador quiere aprender: scanf, gets y getc

Al igual que `printf()`, la función `scanf()` pertenece a `stdio`, la librería de entrada y salida estándar de C. Mediante `scanf()` seremos capaces de solicitar información al usuario, de manera que el sistema pueda almacenarla en una variable del tipo adecuado.

El cometido de esta función es leer datos formateados que proceden de la entrada estándar o stdin. A menudo, el usuario será el encargado de alimentar dicha entrada mediante el teclado.

La sintaxis de `scanf()` –abreviatura de *scan format*: analizar con formato– es la siguiente:



**Recuerda 2.4: Sintaxis scanf()**

```
resultado=scanf(formato, lista de argumentos separados por comas);
```

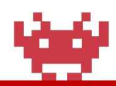
Vamos a explicar con más detalle dicha sintaxis a través de los argumentos recibidos y del resultado devuelto por `scanf()`:

- El primer argumento es una cadena de texto entrecomillada. Su cometido es indicar el formato de los datos que la función espera analizar del flujo de entrada estándar, de manera similar al argumento que especificaba el formato en la función `printf()`. En la cadena de texto se colocarán tantos especificadores de formato como datos individuales se deseen solicitar mediante la entrada estándar. En el caso de la entrada por teclado, el usuario separará cada introducción de dato mediante espacios en blanco, tabulaciones o saltos de línea.

- El segundo argumento es, a su vez, una lista de argumentos separados por comas. Cada argumento almacenará los valores introducidos por stdin en riguroso orden

de entrada. Hay que tener en cuenta que los argumentos de almacenamiento de valores deben ser direcciones de memoria. Para indicar la dirección de una variable se utilizará el operador ampersand (`&`). En lenguaje C, el operador `&` hace referencia a la ubicación en memoria de la variable a la que acompañe.

- El resultado que la función devuelve nos va a servir para comprobar si `scanf()` ha llevado a cabo el análisis del flujo de entrada con éxito. Así, la función retornará el número de datos que se han leído de manera correcta. Si devuelve 0 significará que no se ha podido leer ningún valor. Recoger el resultado en una variable es totalmente opcional para el programador.

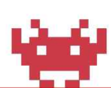


**Ejemplo 2.5: Introduzca un número**

En pantalla aparecerá el mensaje “Introduzca un número entero:”. Se insta al usuario a que teclee un número. Cuando el usuario ejecute dicha acción y pulse INTRO, el programa almacenará dicho valor en la variable entero, ya que en la función scanf() se le ha pasado la dirección de memoria de dicha variable, utilizando el operador &.

**Código**

```
#include <stdio.h>
main()
{
    int entero;
    printf("Introduzca un numero entero: \n");
    scanf("%d", &entero);
}
```



**Ejemplo 2.6: Un educado computador**

Este ejemplo hará que el ordenador demuestre sus buenos modales. Saludará al usuario y le preguntará su nombre. Una vez que el usuario introduzca por teclado la información solicitada, el programa volverá a saludarlo, dirigiéndose en esta ocasión a su interlocutor por su nombre de pila.

**Código**

```
#include <stdio.h>
main()
```

```
{
    char nombre[15];
    printf ("Buenas tardes. Como te llamas? \n");
    scanf("%s", nombre);
    printf("Encantado de conocerte, %s", nombre);
}
```

Examinemos con más detenimiento el ejemplo del saludo. Es necesario tener en cuenta lo particular del tratamiento de cadenas de texto en C. Recordemos que dicho lenguaje no dispone de un tipo de datos específico cuya finalidad sea gestionar cadenas. Así, para almacenar un nombre, una frase o cualquier otro texto, utilizaremos arrays o vectores de caracteres. En el ejemplo hemos creado una variable llamada nombre cuyo tipo es un `char[]`, y su tamaño, 15.

Al avispado programador le llamará la atención el uso de `scanf()` en el ejemplo, puesto que en el segundo argumento hemos omitido el operador `&`. En este caso lo hemos hecho de forma deliberada, puesto que la variable nombre es de tipo array, y esta clase de variables apunta de manera directa a una dirección de memoria, de manera que se evita la necesidad de tener que utilizar el carácter ampersand a la hora de referenciarla en la función `scanf()`.

Si extendemos el ejemplo haciendo que preguntemos nombre y apellidos de forma secuencial, puede que nos topemos con un problema imprevisto que nos servirá para introducir un método más fiable que `scanf()` para almacenar cadenas de caracteres desde entrada estándar. Primero, veamos el código que a priori se nos ocurriría con lo estudiado hasta el momento.



**Ejemplo 2.7: Código erróneo**

Un ejemplo que no funcionará demasiado bien.

**Código**

```
#include <stdio.h>
main()
{
    char nombre[15];
    char apellido[15];
    printf("Me podria decir su nombre? \n");
    scanf("%s", nombre);
    printf("Seria tan amable de indicarme su primer
        apellido? \n");
    scanf("%s", apellido);
```

```
    printf("Encantado de conocerle, %s %s", nombre,
        apellido);
}
```

Al compilar y ejecutar este código, es posible que todo vaya sobre ruedas. Si a la primera pregunta del ordenador introducimos "Juan" y a la segunda respondemos con "Pérez", el sistema nos saludará como si fuera una mañana soleada y alegre. Sin embargo, si nuestro nombre es compuesto, los nubarrones ennegrecerán el cielo, haciendo que la pregunta del apellido salte por los aires y no se nos otorgue la oportunidad de que sea respondida. ¿Por qué ocurre esto?

```
Me podria decir su nombre?
Juan Carlos
Seria tan amable de indicarme su primer apellido?

Encantado de conocerle, Juan Carlos
```

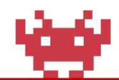
La respuesta está en el búfer de entrada, que al ser gestionado mediante **`scanf()`** opera de la siguiente manera: al encontrarse con un espacio en blanco, lo deja en el búfer. Esto provoca que el siguiente **`scanf()`** recepcione y se " trague" el contenido de dicho búfer, creyendo que dicho espacio en blanco es la respuesta que hemos introducido a la segunda pregunta. Para solucionar este tipo de situaciones y gestionar la entrada de cadenas complejas que incluyan espacios, se utiliza la función **`gets()`**, que nos permitirá asignar cadenas de texto a una variable de tipo vector de caracteres a través de una sintaxis muy sencilla:



**Recuerda 2.5: Sintaxis gets()**

```
gets(variable);
```

En general, conviene utilizar **`gets`** siempre que exista la posibilidad de que la entrada de texto por parte del usuario tenga más de una palabra. Por lo tanto, si recodificamos nuestro ejemplo del saludo con nombre y apellido con dicha función, nuestra mañana seguirá siendo feliz.



**Ejemplo 2.8: Código OK**

Modificación del ejemplo 2.7 para que los buenos modales del ordenador terminen con final feliz.

**Código**

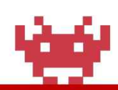
```
#include <stdio.h>
main()
{
    char nombre[15];
    char apellido[15];
    printf("Me podria decir su nombre?");
    gets(nombre);
    printf("Seria tan amable de indicarme su primer
        apellido? \n");
    gets(apellido);
    printf("Encantado de conocerle, %s %s", nombre,
        apellido);
}
```

Por último, y de forma complementaria a la función `putc()` que aprendimos en el apartado anterior, también existe la función `getc()` para recoger un carácter del flujo de entrada especificado por parámetro. Su sintaxis es:

**Recuerda 2.6: Sintaxis getc()**

```
getc(entrada);
```

A continuación, incluimos un ejemplo que repasa varias de las funciones de entrada y salida de texto y caracteres:



**Ejemplo 2.9: Ejemplo de repaso completo**

Un ejemplo que repasará las funciones de entrada y salida de texto y caracteres repasadas hasta el momento

**Código**

```
#include <stdio.h>
main()
{
    char c;
    char nombre[20];
```

```
    puts("Introduzca su nombre real");
    gets(nombre);
    puts("Introduzca su nombre en clave mediante un
        unico caracter");
    c = getc(stdin);
    puts("Su nombre real es...");
    puts(nombre);
    puts("Su nombre en clave es...");
    putc(c, stdout);
}
```

## 2.4. Matemagia: jugando con los números

La Matemagia es un tipo de ilusionismo que se basa en la utilización de métodos matemáticos que, en apariencia, son capaces de distraer al espectador, haciéndole creer que el mago en cuestión es capaz de adquirir un conocimiento que, en teoría, únicamente posee la persona o personas que asisten al truco.

Si analizamos detenidamente lo que propone la Matemagia, podremos concluir que esta curiosa disciplina suele cumplir dos importantes premisas. Por un lado, los ejercicios matemágicos permiten que acometamos la enseñanza y el aprendizaje de los conceptos matemáticos desde un punto de vista lúdico. Por otro, hay que señalar que la amplia mayoría de ejercicios de este tipo siguen una serie de pasos ordenados, haciendo que estén al alcance de cualquiera. Aprovechemos esta última premisa para colocar una chistera y dotar de varita mágica a un algoritmo.

El truco es el siguiente: el ordenador se convertirá en un mentalista capacitado para adivinar cualquier número que esté pensando el usuario. El cibernético prestidigitador solicitará la colaboración del voluntario de turno -que se encuentra al otro lado del teclado-, pidiéndole que escoja un número cualquiera. El usuario lo mantendrá en su mente sin revelarlo a nadie. A continuación, el ordenador irá solicitando que el voluntario realice una serie de operaciones aritméticas, un proceso que desembocará en una cifra. Finalmente, y tras pedir dicho resultado al usuario, el ordenador desvelará el número secreto, sabiendo con total seguridad que tiene garantizado el éxito total en su adivinación.

Una vez explicada la mecánica, vamos a desenmascarar al mago, revisando el algoritmo que necesitaremos para realizar este interesante truco de prestidigitación matemática:

- Imprimir un mensaje que anime al jugador a pensar un número al azar.
- Imprimir un mensaje que ordene al jugador realizar -de forma secuencial- las siguientes operaciones con el número secreto:

- Doblar el número secreto.
- Multiplicar por cinco el resultado de la operación del paso anterior.
- Solicitar al jugador que introduzca el valor obtenido en la operación del paso anterior.

- Tomar la cifra introducida por el jugador y quedarse con la cifra al completo excepto el último dígito.
- Imprimir un mensaje con la cifra obtenida en el paso anterior y anunciar que dicha cifra es el número que el jugador pensaba.



**Juego 2.1: Matemagia: jugando con los números**

Se le pide al lector que codifique un programa en lenguaje C que implemente el algoritmo matemágico que acabamos de describir.

**Código**

```
#include <stdio.h>

main()
{
    /* Variables de control */
    int resultado;
    int secreto;

    /* Presentacion */
    printf("Voy a demostrarte un truco de magia,
        adivinando cualquier numero que puedas pensar
        de seis cifras \n");
    printf("Asi, lo primero que voy a pedirte es que
        escojas un numero y lo memorices...\n");

    /* Operaciones interactivas */
    printf("Bien... ahora, dobla tu número secreto y
        guarda el resultado en tu mente\n");
    printf("Lo siguiente que debes hacer sera
        multiplicar por cinco el resultado de la
        operacion anterior\n");
    printf("¿Ya? Bien, podrias escribir el resultado de
         dicha multiplicación?\n");
    scanf("%d", &resultado);

    /* Obtención del número secreto */
```

```
    secreto = resultado / 10;

    /* El prestigio: ejecución final del truco */
    printf("Hemos llegado al final. Ah, se me olvidaba.
        El número secreto que pensaste es el... %d \n"
        , secreto);
    printf("Gracias y recuerda que un buen mago nunca
        revela sus trucos");
}
```

Fuente del truco de magia: Guía del profesor matemático, Bradley Fields, `http://mathemagic.com`.

## 2.5. Bifurcación de caminos: los comandos if / else

Aunque en el próximo capítulo estudiaremos este tema con detalle, introduciremos ahora la pareja de comandos formada por `if` y `else`, que constituyen la vía más utilizada para controlar el flujo de un proceso dentro de la programación estructurada. Si tuviéramos que enunciar una metáfora que se acercara lo más posible al funcionamiento de tales comandos, apostaríamos por un paseante que llega a una bifurcación de caminos. Nuestro paseante debe decidir cuál de los dos caminos tomar, sabiendo que, tras recorrer uno u otro, su camino confluirá en un mismo punto al que ambas rutas llegarán cuando finalicen cada uno de los tramos.

Si examinamos con detenimiento nuestra particular comparativa, hay algo que aún no hemos explicado. ¿Qué es lo que mueve al caminante a escoger una u otra ruta? La decisión sería lo que, informáticamente hablando, denominaremos condición, la cual debe ser evaluada por el propio programa. Así, si la condición se cumple, el programa escogerá la primera rama de la bifurcación, ejecutando las instrucciones que se encuentren en dicho bloque, mientras que en caso contrario, el flujo virará hacia la segunda rama, haciendo lo propio con las instrucciones allí colocadas. La sintaxis de if / else en Lenguaje C es la siguiente:

**Recuerda 2.7: Sintaxis if/else**

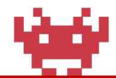
```
if ( condición )
{
   bloque de instrucciones ruta 1
}
```

```
else
{
   bloque de instrucciones ruta 2
}
```

Lo que denominamos condición no es más que una expresión construida con operadores lógicos y/o relacionales, de modo que su resultado a la hora de ser evaluada sea verdadero o falso. Cuando la evaluación devuelve un número distinto de cero, el lenguaje interpreta que es verdadero (true), con lo cual aseveraremos que la condición se cumple. Para construir la expresión a evaluar tendremos a nuestra disposición una serie de operadores. A continuación, detallaremos los más comunes.

- Operadores relacionales:
  - ==: Operador "igual".
  - !=: Operador "distinto a".
  - >: Operador "mayor que".
  - <: Operador "menor que".
  - >=: Operador "mayor o igual que".
  - <=: Operador "menor o igual que".
- Operadores lógicos:
  - &&: Operador de conjunción o "and". Evalúa si dos operandos son verdaderos *a la vez*.
  - ||: Operador de disyunción o "or". Evalúa si al menos uno de dos operandos es verdadero.

Por su parte, los bloques de instrucciones se delimitan con llaves, aunque estas serán opcionales en el caso de que el bloque en cuestión se componga de una única instrucción. Veamos algunos ejemplos.

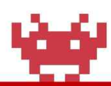


**Ejemplo 2.10: Mayor o menor**

En este ejemplo, el programa pregunta por pantalla al usuario por un número cualquiera. La condición que rige la bifurcación de caminos determina si el número almacenado en la variable i es mayor que 5 (i >5). Así, si el número introducido es mayor que 5, el programa imprimirá por pantalla la frase "Tu número es mayor que 5". En otro caso, el programa hará lo propio con la cadena de texto "Tu número es igual o menor que 5". Finalmente, observad que se han utilizado las llaves aunque su uso sería opcional en el ejemplo.

**Código**

```
#include <stdio.h>
main()
{
    int i;
    printf("Introduce un numero");
    scanf("%d", &i);
    if (i > 5)
    {
        printf("Tu numero es mayor que 5");
    }
    else
    {
        printf("Tu numero es igual o menor que 5");
    }
}
```



**Ejemplo 2.11: Nuestra primera aventura**

¡Comienza la aventura! El usuario es un intrépido explorador en cuyo camino se topa un insidioso troll. ¿Qué hacer? El programa otorgará dos opciones bien distintas. Cada opción está numerada. El usuario debe decidir e introducir por teclado su movimiento. Si es tan torpe como para introducir cualquier entrada distinta de 1 y de 2, el programa lo castigará convirtiéndolo en una suculenta cena para el troll.

Como vemos, el lenguaje C nos da la posibilidad de anidar condiciones. Así, la condición anidada, una vez que queda claro que la opción introducida debe ser 1 o 2, será evaluar en qué caso concreto nos encontramos. Dependiendo de si hemos sido impulsivos o reflexivos, nuestra carrera como aventureros acabará de golpe o podrá continuar, esta vez con un nuevo amigo, embelesado por nuestras dotes como solistas.

**Código**

```
#include <stdio.h>

main()
{
    int opcion;
```

```
    printf("Un troll te impide el paso. Selecciona tu movimiento:\n");
    printf("1 - Empujarlo hacia el abismo \n");
    printf("2 - Cantar una tonadilla infantil \n");
    scanf("%d",&opcion);

    if (opcion!=1 && opcion!=2)
       printf("Movimiento ilegal: El troll te devora");
    else
       if (opcion==1)
          printf("Tropiezas, cayendo al abismo. Game over.");
       else
          printf("El troll se ablanda y entabla amistad contigo");
}
```

## 2.6. Preguntas y respuestas

En Diciembre de 1979, dos periodistas se encontraban a la lumbre de un salón cualquiera, a salvo del frío que reinaba en el crudo invierno canadiense. Hasta arriba de cerveza, Scott Abbott y Chris Haney intentaban matar el tiempo a través de un tablero de Scrabble al que le faltaban unas cuantas fichas de letras. Viendo que aquel juego de mesa se les quedaba cojo, decidieron inventar uno propio. Así, ebrios de alcohol y creatividad, diseñaron las líneas maestras de lo que se convertiría en todo un fenómeno social: el Trivial Pursuit. Dos años después se comercializaba la primera edición del juego. Hoy día sería complicado encontrar a una persona que no haya jugado al menos una partida de este desafío cultural de preguntas y respuestas.



**Juego 2.2: Trivial Pursuit**

En el presente ejercicio, no vamos a ser tan exigentes como para crear las mecánicas base del juego de Abbott y Haney. En cambio, solicitaremos al lector que componga un programa en el que el ordenador realice tres preguntas al jugador. Cada pregunta deberá ir acompañada de tres posibles respuestas, siendo una de ellas la correcta. El programa debe ser capaz de hacer que se sume un punto al marcador del jugador por cada respuesta correcta. Al finalizar la ronda de juego, el programa desvelará al concursante su puntuación final.

**Código**

```c
#include <stdio.h>

int main()
{
    /* Variables de control */
    int opcion;
    int marcador = 0;

    /* Presentación */
    printf("Bienvenido a nuestro juego de preguntas y respuestas\n");
    printf("Demuestra tu cultura general y alcanza un gran marcador
    de puntos\n");

    /* Primera pregunta */
    printf("Atencion, pregunta: Cual es el lema de la casa Stark de
    Invernalia?\n");
    printf("1 - Se acerca el invierno \n");
    printf("2 - Uno para todos y todos para uno \n");
    printf("3 - Los Stark siempre pagan sus deudas\n");
    printf("Elige una opcion introduciendo un numero (1,2 o 3)\n");

    scanf("%d",&opcion);

    if (opcion!=1 && opcion!=2 && opcion!=3)
    {
        /* Opción ilegal */
        printf("No has seguido las reglas del juego. No ganas ningun punto.\n");
    }
    else
    {
        /* Opción legal. Verificar si es la correcta */
        if (opcion==1)
        {
            /* Acierto. Sumar un punto al marcador */
            printf("Enhorabuena. Respuesta correcta.\n");
            marcador++;
        }
```

```
        else
        {
           printf("Respuesta incorrecta. Sigue
              intentándolo\n");
        }
    }

    /* Segunda pregunta */
    printf("Atencion, pregunta: Que apodo tenia el
       autor de la novela Don Quijote de la Mancha?\n"
       );
    printf("1 - El hilarante hidalgo \n");
    printf("2 - El manco de Lepanto \n");
    printf("3 - El potro de Vallecas \n");
    printf("Elige una opcion introduciendo un numero
       (1,2 o 3)");

    scanf("%d",&opcion);

    if (opcion!=1 && opcion!=2 && opcion!=3)
    {
        /* Opción ilegal */
        printf("No has seguido las reglas del juego. No
           ganas ningun
        punto.\n");
    }
    else
    {
        /* Opción legal. Verificar si es la correcta */
        if (opcion==2)
        {
           /* Acierto. Sumar un punto al marcador */
           printf("Enhorabuena. Respuesta correcta.\n");
           marcador++;
        }
        else
        {
           printf("Respuesta incorrecta. Sigue
              intentandolo\n");
        }
    }

    /* Tercera pregunta */
```

```
    printf("Atencion, pregunta: Con cual de estos
        comandos podemos controlar el flujo de un
        programa en C?\n");
    printf("1 - if / else \n");
    printf("2 - printf() \n");
    printf("3 - scanf() \n");
    printf("Elige una opcion introduciendo un numero
        (1,2 o 3)");

    scanf("%d",&opcion);

    if (opcion!=1 && opcion!=2 && opcion!=3)
    {
        /* Opción ilegal */
        printf("No has seguido las reglas del juego. No
            ganas ningun punto.\n");
    }
    else
    {
        /* Opción legal. Verificar si es la correcta */
        if (opcion==1)
        {
            /* Acierto. Sumar un punto al marcador */
            printf("Enhorabuena. Respuesta correcta.\n");
            marcador++;
        }
        else
        {
            printf("Respuesta incorrecta. \n");
        }
    }

    /* Marcador final */
    printf("Fin de la partida. Tu puntuación final ha
        sido de %d", marcador);
}
```

Fuente de la historia del Trivial Pursuit: `http://www.todayifoundout.com/index.php/2014/09/brief-history-trivial-pursuit/`.

## 2.7. Decidir entre múltiples opciones: el comando switch

Retomemos nuestra idea original de la bifurcación de caminos y subamos la apuesta a lo que sería una auténtica encrucijada. Múltiples rutas y una sola elección basada en el valor de una expresión simple. Por ejemplo, imaginemos que nos sentamos en la mesa de un restaurante y el camarero nos ofrece la carta del menú. Tenemos una multitud de platos a elegir, pero solo vamos a poder escoger uno de ellos, y dependiendo de nuestra decisión, el camarero realizará una serie de acciones condicionadas al plato escogido. Para implementar esta situación real en el universo digital del lenguaje C optaremos por el comando `switch`.

Antes de explicar la sintaxis de `switch`, hagamos la siguiente observación: a la vista de los ejemplos y ejercicios realizados en el punto anterior, el programador estaría en condiciones de implementar un programa que simulara la elección del comensal ante el menú ofrecido por el camarero. Bastaría con que utilizara el anidamiento de estructuras condicionales con `if` / `else`. Sin embargo, tal y como hemos adelantado, C dispone de un comando que le va a facilitar las cosas, ahorrándole una buena cantidad de tiempo, esfuerzo y líneas de código. Se trata del comando `switch`, el cual presenta la siguiente sintaxis:

**Recuerda 2.8: Sintaxis switch()**

```
switch ( expresión )
{
   case valor constante 1:
      bloque de instrucciones ruta 1;
     break;

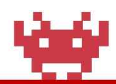
   case valor constante 2:
      bloque de instrucciones ruta 1;
     break;
   /* … */
   default:
   bloque de instrucciones ruta por defecto;
}
```

La estructura de control de flujo `switch` / `case` basa su mecanismo en una expresión cuyo valor será utilizado por el programa para poder decidir en consecuencia qué camino tomar. Normalmente, dicha expresión suele ser una variable de tipo `int` o `char`, aunque también se admiten otros tipos de valores numéricos enteros tales como `short`, `long`, `signed` o `unsigned`.

Tras realizar la evaluación de la variable encerrada entre paréntesis que acompaña al comando **switch**, el programador definirá una serie de opciones posibles respecto al valor de la mencionada variable. Cada opción se define mediante un comando case acompañado del valor en cuestión.

Si el valor del **case** en cuestión encaja con el valor de la variable evaluada, el flujo del programa ejecuta el bloque de instrucciones definido a continuación de dicho **case**. Sin embargo, en este punto hay que tener en cuenta que el programa seguirá ejecutando el resto de case definidos a continuación del case en cuestión, a menos que el programador rompa con dicho flujo y se salte el resto de casos definidos en el **switch**. Esta "rotura" de flujo se realiza mediante el comando **break**, que debemos colocar justo al final de cada bloque de instrucciones en todos y cada uno de los case definidos. El comando break es opcional, pero su utilización se nos antoja imprescindible para garantizar el buen funcionamiento de la estructura de control.

Por último, si queremos que el programa ejecute un bloque de instrucciones en el caso de que ninguno de los case definidos encaje con el valor evaluado en el **switch**, tendremos que hacer uso del comando **default**, que irá colocado justo después del último **case** definido. El uso de este comando "por defecto" será totalmente opcional, aunque una vez más se recomienda como buena práctica.



**Ejemplo 2.12: ¡A comer!**

Vamos a intentar asentar toda esta teoría a través de la práctica mediante el siguiente ejemplo. Un hambriento comensal debe seleccionar el número de plato que desea. Así, el ordenador solicita por pantalla al usuario que introduzca dicha cifra mediante la entrada estándar –recordemos, por defecto, que la entrada estándar es el propio teclado–.

**Código**

```
#include <stdio.h>
main()
{
    int i;
    printf("Buenas noches! Soy el camarero del
        restaurante. Aquí
    tiene la carta de menu para esta noche:\n");
    printf("1. Pato a la naranja\n");
    printf("2. Bistec de buey a las finas hierbas\n");
    printf("3. Tortilla española\n");
    printf("4. Caballa con piriñaca\n");
    printf("Por favor, escoja el número del plato que
        desea para su cena\n");
```

```

    scanf("%d", &i);

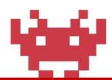
    switch (i)
    {
        case 1:
        printf ("Marchando un pato a la naranja \n");
        case 2:
        printf ("Marchando un buen bistec \n");
        case 3:
        printf ("Marchando esa tortilla \n");
        case 4:
        printf ("Marchando una caballa gaditana \n");
        default:
        printf ("Lo siento, caballero. Ese plato no lo
            tenemos. Me temo que pasara un poco de
            hambre \n");
    }
}
```

Aquí tenemos, por fin, el menú del restaurante trasladado a lenguaje C. La condición que rige la selección de caminos es el valor de la variable `i`, que es precisamente el lugar en el que se ha almacenado el número tecleado por el usuario. De esta manera, el programa imprimirá la comanda de cocina en función del plato escogido. En el caso de que el usuario introduzca un número que no conste en el menú, el comensal se quedará sin cena.

Pero... ¿es correcto el código del programa que hemos propuesto? Si lo revisamos con atención, veremos que hay un detalle muy importante que no hemos considerado. Basta con ejecutar el programa y comprobar la respuesta por pantalla al elegir un suculento pato a la naranja...

```
Marchando un pato a la naranja
Marchando un buen bistec
Marchando esa tortilla
Marchando una caballa gaditana
Lo siento, señor. Ese plato no lo tenemos. Me temo que pasara un poco
de hambre
```

A la hora de confeccionar el código del programa hemos cometido un error que ha provocado que nuestro pobre comensal vaya a sufrir una auténtica indigestión. En concreto, hemos olvidado "romper" con el flujo del programa con break al finalizar cada case. Gracias a este descuido, la ejecución del código realizará todas las acciones sucesivas al case con el que coincida el valor introducido.



**Ejemplo 2.13: ¡A comer con moderación!**

A continuación, incluimos el programa corregido y revisado:

**Código**

```
#include <stdio.h>
main()
{
    int i;
    printf("Buenas noches! Soy el camarero del
        restaurante. Aquí
    tiene la carta de menu para esta noche:\n");
    printf("1. Pato a la naranja\n");
    printf("2. Bistec de buey a las finas hierbas\n");
    printf("3. Tortilla española\n");
    printf("4. Caballa con piriñaca \n ");
    printf("Por favor, escoja el numero del plato que
        desea para su cena \n");

    scanf("%d", &i);

    switch (i)
    {
        case 1:
        printf ("Marchando un pato a la naranja \n");
        break;
        case 2:
        printf ("Marchando un buen bistec \n");
        break;
        case 3:
        printf ("Marchando esa tortilla \n");
        break;
        case 4:
        printf ("Marchando una caballa gaditana \n");
        break;
        default:
        printf ("Lo siento, señor. Ese plato no lo
            tenemos. Me temo que pasara un poco de
            hambre \n");
    }
}
```

## 2.8. Juego de preguntas: nueva versión

En la sección anterior creamos nuestra particular versión del Trivial. En el ejercicio que proponemos a continuación, pediremos al programador que rehaga el mismo juego de preguntas y respuestas utilizando en esta ocasión la instrucción switch para evitar el anidamiento de estructuras condicionales if/else.



**Juego 2.3: Trivial Pursuit 2.0**

La solución que proponemos, siguiendo la misma batería de preguntas del ejercicio previamente reseñado, sería la siguiente.

**Código**

```
#include <stdio.h>

main()
{
    /* Variables de control */
    int opcion;
    int marcador = 0;

    /* Presentación */
    printf("Bienvenido a nuestro juego de preguntas y respuestas\n");
    printf("Demuestra tu cultura general y alcanza un gran marcador de puntos\n");

    /* Primera pregunta */
    printf("Atencion, pregunta: Cual es el lema de la casa Stark de Invernalia?\n");
    printf("1 - Se acerca el invierno \n");
    printf("2 - Uno para todos y todos para uno \n");
    printf("3 - Los Stark siempre pagan sus deudas\n");
    printf("Elige una opcion  introduciendo el numero (1,2 o 3)\n");

    scanf("%d",&opcion);

    switch (opcion)
    {
        case 1:
        /* Acierto. Sumar un punto al marcador */
```

```
        printf("Enhorabuena. Respuesta correcta.\n");
        marcador++;
        break;
        case 2:
        case 3:
        printf ("Respuesta incorrecta. Sigue
            intentándolo \n");
        break;
        default:
        printf ("No has seguido las reglas del juego. No
              ganas ningún punto \n");
    }

    /* Segunda pregunta */
    printf("Atencion, pregunta: Que apodo tenia el
        autor de la novela Don Quijote de la Mancha?\n"
        );
    printf("1 - El hilarante hidalgo \n");
    printf("2 - El manco de Lepanto \n");
    printf("3 - El potro de Vallecas \n");
    printf("Elige una opcion introduciendo el numero
        (1,2 o 3)");

    scanf("%d",&opcion);

    switch (opcion)
    {
        case 2:
        /* Acierto. Sumar un punto al marcador */
        printf("Enhorabuena. Respuesta correcta.\n");
        marcador++;
        break;
        case 1:
        case 3:
        printf ("Respuesta incorrecta. Sigue
            intentandolo. \n");
        break;
        default:
        printf ("No has seguido las reglas del juego. No
              ganas ningun punto \n");
    }

    /* Tercera pregunta */
```

```
    printf("Atencion, pregunta: Con cual de estos
        comandos podemos controlar el flujo de un
        programa en C?\n");
    printf("1 - if / else \n");
    printf("2 - printf() \n");
    printf("3 - scanf() \n");
    printf("Elige una opcion  introduciendo el numero
        (1,2 o 3)");

    scanf("%d",&opcion);

    switch (opcion)
    {
        case 1:
        /* Acierto. Sumar un punto al marcador */
        printf("Enhorabuena. Respuesta correcta.\n");
        marcador++;
        break;
        case 2:
        case 3:
        printf ("Respuesta incorrecta. Sigue
            intentándolo \n");
        break;
        default:
        printf ("No has seguido las reglas del juego. No
             ganas ningún punto \n");
    }

    /* Marcador final */
    printf("Fin de la partida. Tu puntuación final ha
        sido de %d", marcador);
}
```

Al revisar la solución propuesta podremos comprobar cómo la parte de código que gestiona la comprobación de cada respuesta proporcionada por el jugador queda mucho más limpia y concisa al evitar la anidación de sentencias condicionales. Por lo demás, el funcionamiento del programa es absolutamente idéntico al del ejercicio de la sección previa.

## 2.9. Ejercicios propuestos

Finalmente os proponemos los siguientes retos de programación. A ver cuántos sois capaces de superar.



**Ejercicio 2.1: Las porciones justas de pizza**

Siempre que se pide pizza en compañía hay alguien que come más que el resto. Para evitar esta grave injusticia se debe realizar un programa que pregunte al usuario los siguientes datos: el radio de la pizza en centímetros, el número de comensales y el número de porciones en los que se pretende dividir la pizza. El programa debe proporcionar la siguiente información por pantalla: a) área de la pizza en centímetros cuadrados; y b) área que corresponde a cada comensal expresada en porciones y centímetros cuadrados.



**Ejercicio 2.2: El vengador pizzero**

Modifique el programa del ejercicio 2.1 para que, después de mostrar los resultados por pantalla, pregunte al usuario por el número de porciones engullidas por el comensal que más comió. Asumiendo que cada porción tiene aproximadamente 250 calorías y sabiendo que cada minuto de *footing* quema 11 calorías, indique al comensal glotón cuantos minutos debe correr para quemar las calorías adicionales que ganó por comer más que la media.

# Capítulo 3

# Programación estructurada (y más)

En el capítulo anterior se han presentado algunas nociones básicas de C, suficientes para desarrollar, por ejemplo, juegos de preguntas y respuestas. Los programas realizados hasta el momento han sido relativamente sencillos y unas decenas de líneas de código han sido suficientes para completarlos. Salvo que vuestro mayor deseo sea el de programar una versión computerizada del popular Trivial Pursuit, os habréis quedado con ganas de más. Teniendo en cuenta que no es extraño que un programa comercial tenga centenares, miles o incluso millones de líneas de código, cabría preguntarse si existe alguna forma de programar que mejore la probabilidad de éxito de tamaña empresa. Por fortuna, la respuesta a esta pregunta es afirmativa. Todo problema de programación puede ser resuelto utilizando únicamente tres tipos de estructuras que se estudian en este capítulo.

El objetivo de este tema es afianzar y ampliar vuestros conocimientos de C, y enseñaros a desarrollar programas de forma estructurada, lo que resulta más fácil de decir que de hacer. De la misma forma que el alfarero debe adquirir unas nociones técnicas elementales antes de realizar un jarrón, aprenderéis cuáles son las estructuras básicas de programación que se utilizan en C. El tiempo y la práctica os permitirán interiorizarlas y solo así llegaréis a convertiros en grandes programadores.

## 3.1. El flujo del programa: be water, my friend

El primer protagonista de este capítulo tiene nombre propio: el flujo del programa. Detrás de este concepto se esconde una idea que explicaremos con una metáfora: imaginad por un momento un río. Si os acercáis bien al mismo apreciaréis cómo el agua fluye con una dirección bien determinada. Imaginemos que hacemos un barquito de papel y lo ponemos a navegar. No cabe duda de que este seguirá la dirección de la corriente. Con todo, la travesía del barquito no estará desprovista de incertidumbres y peligros. Ocasionalmente el barco encontrará bifurcaciones en el río y tomará un camino u otro en función de sus propias circunstancias. Asimismo, a veces quedará

atrapado en remolinos que lo obligarán a navegar una y otra vez una misma parte del río hasta que por fin lo dejen marchar.

Pensemos en la metáfora anterior desde una perspectiva informática. En este contexto el barco es la CPU y el río está formado por las instrucciones del programa. Cada tramo del río sobre el que el barco navega es una instrucción que es ejecutada por la CPU. De este modo, la trayectoria que el barco sigue por el río representa el *flujo* que sigue el programa en su ejecución, el cual determina el orden en que se llevan a cabo las instrucciones que lo componen. Es decir, el barquito-CPU surca el río de las instrucciones siguiendo una trayectoria a la que llamamos flujo del programa.

Volviendo al mundo informático, conviene precisar que el flujo del programa hace referencia a la manera en que las instrucciones fluyen una tras otra a través del procesador, es decir, son ellas las que se mueven una tras otra hasta llegar a la CPU y ser ejecutadas. Este, y no otro, es el significado de flujo del programa, aunque esta perspectiva es mucho menos visual que la de nuestro barco CPU. La tarea del programador es colocar sabiamente las instrucciones para que la CPU las ejecute de la manera deseada. Al fin y al cabo, un ordenador puede no comportarse como queremos pero siempre lo hace como le decimos.

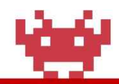
Una vez introducido este concepto, ya estamos preparados para conocer los tipos de estructura que componen un buen programa en C.

## 3.2. Estructura secuencial

Es el caso más sencillo de estructura. Para que un programa sea secuencial simplemente se requiere que las instrucciones se ejecuten una tras otra de modo que no comience a ejecutarse una nueva instrucción hasta que no haya terminado de hacerlo la que se está ejecutando en ese preciso instante.

Programar de forma secuencial en C se hace de manera natural. Las instrucciones se separan entre sí mediante el carácter punto y coma (“;”), que delimita dónde empieza y termina cada una de ellas. Una vez termina de ejecutarse una instrucción se continúa con la siguiente salvo que el programador introduzca algo que altere el flujo natural del programa.

Veamos un sencillo ejemplo de código secuencial.



**Ejemplo 3.1: Código secuencial**

Este ejemplo es similar a los que ya se introdujeron en el capítulo anterior e ilustra el orden natural que sigue el flujo del programa en C. El siguiente programa pide al usuario que introduzca un número y lo muestra por pantalla.

**Código**

```
#include <stdio.h>
int main()
{
    int num;
    printf("Introduzca un numero\n");
    scanf("%d",&num);
    printf("\nEl numero fue %d\n",num);
}
```

El programa siempre empieza a fluir a partir de su primera instrucción, que no es otra que la primera instrucción de la función principal o *main*. Una vez colocado nuestro barquito-procesador al principio de dicha función se puede predecir la trayectoria que seguirá. En este sencillo ejemplo el barquito-procesador fluye por el río de instrucciones de la siguiente forma:

1. Empieza en la línea `int num`, así que crea espacio en memoria para almacenar una variable de tipo entero llamada `num`.
2. Una vez se ha terminado de navegar por la zona de `int num` el barquito-procesador llega a la siguiente instrucción. Se imprime por pantalla el mensaje: `Introduzca un numero`.
3. A continuación, se espera a que el usuario introduzca un número por teclado. El dato introducido se guarda en la dirección de memoria en que se almacena la variable `num`, denotado por `&num`.
4. Justo después se imprime por pantalla el siguiente mensaje: `El numero fue:` y a continuación se añade el contenido de la variable `num`.
5. Finalmente el programa termina.

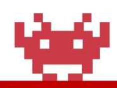
El ejercicio de calcular *mentalmente* el flujo que sigue el programa es muy útil, especialmente para detectar si el flujo transcurre como se desea. Si no fuera así es que algo va mal en el código y habrá que reescribir o añadir alguna instrucción.

## 3.3. Estructura condicional

El segundo tipo de estructura fue introducido en el capítulo anterior. Se trata de la estructura de control condicional `if/else` y su versión de elección múltiple `switch`. Desde la perspectiva del flujo del programa, la estructura condicional genera un punto de bifurcación en el que el recorrido del barquito-CPU puede seguir una de diferentes

rutas alternativas que convergen en un mismo punto. Así, cuando se cumplen una serie de condiciones el barco sigue por un camino; si no, continúa por otro.

Por ejemplo, con la instrucciones `if` y `else` se generan dos rutas alternativas. Nótese que de las dos palabras solamente la primera (`if`) es obligatoria, ya que es la que plantea la bifurcación como tal. Veamos un sencillo ejemplo con un programa que pide un número al usuario y le dice si este es par o impar.



**Ejemplo 3.2: Ejemplo de bifurcación de flujo**

Este ejemplo muestra cómo el flujo puede seguir uno de dos caminos posibles.

**Código**

```
#include <stdio.h>
 int main()
 {
     int num;
     printf("Introduzca un numero\n");
     scanf("%d",&num);
     if(num%2==0)  //si el resto de num/2 es igual a 0
     {
         printf("\nEl numero es par\n");
     }
     else
     {
         printf("\nEl numero es impar\n");
     }
 }
```

Realicemos el ejercicio mental de seguir el flujo del programa. Como siempre, empezamos en la primera instrucción de `main`:

1. Se crea en memoria una variable llamada `num`.
2. Se imprime por pantalla un mensaje solicitando al usuario que introduzca un número.
3. Se recoge el dato introducido por el usuario a través del teclado y se almacena en la variable num.
4. A partir de aquí pueden suceder dos cosas:
    - Si el resto de dividir el número entre 2 es 0, el número es par por definición. Por tanto, se imprime en pantalla el mensaje correspondiente.

- En caso contrario, el número es impar, por lo que se imprime "`El numero es impar`" por pantalla.

  Con independencia de cuál sea la ruta escogida (`if` o `else`), el flujo sale de la estructura condicional y ambas ramas vuelven a unirse en una sola.

5. El programa termina.

Como decíamos antes, todas las bifurcaciones de este tipo contienen un `if`, pero no sucede lo mismo con `else`. La ausencia de `else` implica que no se ejecutará ninguna instrucción si no se cumple la condición indicada en `if`. Formalmente sigue habiendo dos rutas alternativas, aunque en una de ellas la CPU no hace nada.

Nótese que las llaves que envuelven a las líneas 9 y 13 del ejemplo 3.2 son opcionales cuando `if` o `else` conducen a la ejecución de una única instrucción. Si se quiere que cualquiera de ellos afecte a un bloque de instrucciones es preciso "empaquetar" con llaves dicho bloque. No obstante, durante vuestros primeros programas es recomendable que incluyáis las llaves siempre. Esta práctica os ahorrará más de un quebradero de cabeza.

A medida que vuestro dominio del lenguaje crezca realizaréis estructuras condicionales cada vez más complejas. De hecho, serán los programas los que os inviten a utilizarlas. Finalmente, la estructura condicional `switch` afecta al flujo del programa de la misma manera que `if/else`, si bien cuenta con la ventaja de poder generar más de dos caminos alternativos en caso de que sea necesario.

## 3.4. Estructura iterativa

Llegamos por fin a una estructura de control de flujo nueva. En ocasiones resulta interesante que el barquito viaje una y otra vez por una misma zona del río de instrucciones. A este tipo de estructura, que podríamos ver como un remolino que lo obligara a repetir parte de su recorrido una y otra vez hasta que consigue escapar, se la conoce estructura iterativa. En el argot de los programadores, se llama *bucle* a la repetición cíclica de una o varias instrucciones en función del cumplimiento de una cierta condición. Asimismo, cada una de estas repeticiones es conocida como *iteración*.

Existen tres formas diferentes de programar bucles en C. Las dos primeras que introduciremos a continuación se utilizan generalmente cuando se desconoce el número de veces que el bucle ha de repetirse. La tercera, en cambio, está indicada especialmente para situaciones en las que este número es conocido.

### 3.4.1. While

La traducción literal de `while` en castellano es mientras. Por tanto, este bucle se repite *mientras* se cumple una condición. Cuando la condición deja de cumplirse, el

bucle deja de repetirse y la ejecución del código continua ejecutándose en las líneas siguientes a `while`.



**Recuerda 3.1: Estructura iterativa while**

`while` repite una serie de instrucciones en bucle *mientras* se cumple una determinada condición que viene indicada entre paréntesis.

**Código**

```
while(condicion)
{
   instrucciones a repetir;
}
```

Cabe indicar que las llaves solo son obligatorias si se desea repetir más de una instrucción. Si solo hay una instrucción a repetir su uso es opcional. Asimismo, la sintaxis utilizada para escribir la condición que determina la repetición del bucle es la misma que se introdujo en el capítulo anterior para `if`.

El siguiente ejemplo ayudará a comprender mejor el uso de `while`.



**Ejemplo 3.3: Número primo**

Se desea realizar un programa que indique si un número introducido por el usuario es o no primo, es decir, para saber si el número puede ser dividido con resto cero por algún número diferente de sí mismo y de la unidad.

**Código**

```c
#include <stdio.h>
int  main()
{
    int numero, divisor=2;
    printf("Introduzca numero mayor o igual a 2:\n");
    scanf("%d", &numero);

    while (numero%divisor != 0)
        divisor=divisor+1;

    if (divisor == numero)
        printf("%d es primo\n", numero);
```

```
    else
        printf("%d no es primo\n", numero);
}
```

Como el ejemplo 3.3 muestra, `while` divide el número introducido por el usuario entre divisores sucesivos empezando por el 2. *Mientras* que el resto de la división sea diferente de 0, el bucle continúa su repetición. La condición deja de cumplirse solo cuando se obtiene un resto igual a 0 en la división. Si el número es primo, esto solo sucede cuando `divisor` es igual a `numero`, de ahí a que esta sea la condición utilizada en el `if` para mostrar un mensaje u otro por pantalla. Si el número no es primo, la condición deja de cumplirse para algún valor de la variable `divisor` inferior a `numero`.

El código anterior tiene también un importante valor didáctico dentro de este capítulo, ya que se trata del primer programa del libro en el que se emplean los tres tipos de estructuras de control de flujo que se utilizan en programación estructurada. En concreto, el flujo se ejecuta secuencialmente y entra en la estructura iterativa `while`. Una vez se rompe la condición de repetición, el flujo continúa y llega inmediatamente a la estructura condicional `if`, momento a partir del cual se bifurca en uno de dos caminos posibles. Ambas alternativas convergen justo a la salida de `if` y conducen a la finalización del programa.

Finalmente, y aunque el ejemplo 3.3 pueda resultar excesivamente matemático, conviene destacar la importancia de los números primos, a los que con justicia se los puede calificar como arquitectos del resto de los números. El hecho de que todo número pueda ser expresado como un producto de números primos es aprovechado por ejemplo por el método de encriptación RSA para garantizar la transmisión segura de información a través de Internet. En particular, dicho algoritmo se basa en que encontrar los factores primos de un número muy grande consume mucho tiempo aún con los ordenadores más potentes. Otra característica fascinante de estos números fue dada en 1742 por Goldbach, quien afirmó que todo número par mayor que 2 puede escribirse como suma de dos números primos. Pese a que se han ofrecido premios millonarios para quien sea capaz de demostrar o refutar esta sencilla afirmación, nadie hasta la fecha ha sido capaz de demostrar de forma general que la conjetura es cierta y tampoco se ha encontrado un número par mayor que 2 para el que no se cumpla. Los responsables de marketing de las empresas también conocen los números primos y lo demuestran cada vez que nos proporcionan un envase con un número primo de unidades. Como sabemos, estos números solo pueden ser divididos por la unidad y por sí mismos, lo que dificulta repartir un número primo de unidades en varias partes iguales, generando con ello un incentivo para comprar más.

### 3.4.2. Do/while

En las películas de acción encontramos con frecuencia dos tipos de personajes, los que preguntan y luego disparan y los que disparan y luego preguntan. Algo similar

sucede con la estructura iterativa **while** y su variante **do/while**. Mientras que **while** comprueba su condición de repetición *antes* de entrar en el bucle, **do/while** lo hace *después* para determinar si debe continuar repitiéndose. Por tanto, las instrucciones que **while** debe repetir pueden no llegar a ejecutarse ni una sola vez. Por ejemplo, en el código del ejemplo 3.3 la instrucción 9 (**div =div+1**) no se ejecuta si **numero** es igual a 2. La estructura **do/while** en cambio siempre ejecuta al menos una vez las instrucciones que encierra, ya que la comprobación de la condición se hace después de la ejecución de las instrucciones y no antes. Por tanto, podría decirse que **while** pregunta y luego dispara mientras que **do/while** dispara y luego pregunta.



**Recuerda 3.2: Estructura iterativa do/while**

**do/while** ejecuta al menos una vez una serie de instrucciones que pueden ser repetidas en bucle *mientras* se cumple una determinada condición que indicada entre paréntesis tras **while**.

**Código**

```
do
{
   instrucciones a repetir;
}
while(condicion);
```

Veamos un ejemplo para ilustrar la utilidad de **do/while**.



**Ejemplo 3.4: Ejemplo de bifurcación de flujo**

Realiza un programa que pida al usuario una contraseña numérica antes de mostrar un mensaje por pantalla.

**Código**

```
#include <stdio.h>
int main()
{
    int clave;
    do
    {
        printf("\n Introduzca la clave: ");
        scanf("%d", &clave);
    }
```

```
    while(clave!=1234);
    printf("Has desbloqueado los secretos de los
        titanes!\"n);
}
```

El ejemplo pide la clave hasta que se introduce 1234. Es evidente que la clave debe ser pedida al menos una vez, lo que hace que **do/while** sea una estructura iterativa más apropiada que **while** para este problema. Obsérvese que el **while** de **do/while** lleva punto y coma para diferenciarlo del **while** que vimos en la sección anterior. Se trata de algo excepcional, puesto que las instrucciones de control de flujo no llevan punto y coma en general.

### 3.4.3. For

Tanto **while** como **do/while** están orientadas para implementar bucles en los que se itera un número de veces que es desconocido a priori para el programador (aunque también podrían ser utilizadas con un número conocido de iteraciones si fuera necesario). En cambio, la estructura de control de flujo iterativa **for** está especialmente diseñada para este tipo de situaciones. Veamos un ejemplo de su utilización para apreciar con claridad esta diferencia.



**Ejemplo 3.5: Ejemplo de bifurcación de flujo**

Un estudiante ha trabajado poco y es castigado a escribir 100 veces "Tengo que estudiar más". Utilice sus conocimientos de C para mitigar el castigo.

**Código**

```
#include <stdio.h>
int main()
{
    int i;
    for( i = 1; i <=100 ; i=i+1)
    {
        printf("%d.- Tengo que estudiar mas \n", i);
    }
}
```

Como puede verse en el ejemplo 3.5, el uso de **for** evita repetir 100 veces la instrucción **printf**. En concreto, el bucle integra en este caso a la variable auxiliar **i**, cuyo valor varía en cada iteración. El valor inicial de dicha variable se asigna en

primer lugar. A continuación, después del punto y coma, se incluye la condición de la que depende que el bucle se siga ejecutando. Finalmente, tras el segundo punto y coma, se indica cómo se actualiza la variable después de cada iteración. Es decir, en este ejemplo la variable `i` comienza con el valor 1 y su valor se incrementa en una unidad cada vez que el bucle completa una iteración, lo que sucede en este caso cada vez que se ejecuta `printf`. Todo lo anterior se repite mientras que se satisfaga la condición `i<=100`. Por tanto, cuando `i` toma el valor 101 la condición de ejecución deja de cumplirse y el bucle termina. Se deja como ejercicio al lector implementar el ejemplo con `while` o `do/while`, que también podrían ser utilizados para ayudar al estudiante.



**Recuerda 3.3: Estructura iterativa while**

El bucle `for` permite integrar en su sintaxis una o más variables cuyo valor inicial y evolución en cada iteración son indicados explícitamente. El bucle se repite mientras se cumple una condición dada.

**Código**

```
for(asignaciones iniciales; condición; actualización variables)
{
   Instrucciones que se repiten;
}
```

Como puede verse, después de `for` aparecen entre paréntesis tres campos separados por punto y coma. El propósito de cada uno es el siguiente:

- Asignaciones iniciales: se definen aquí los valores iniciales de las variables relacionadas con el bucle. De haber más de una variable, se utilizan comas para separar los valores iniciales. (No puede utilizarse aquí ';' ya que es el carácter utilizado para separar los tres campos de `for`.) En caso de que no haya ninguna variable que inicializar puede dejarse un espacio en blanco.

- Condición de ejecución: el bucle se repite mientras que la condición que aquí se especifica se cumpla.

- Actualización de variables: en este campo se definen los cambios que experimentan las variables relacionadas con el bucle después de cada iteración del mismo. Si hay más de una instrucción, deben utilizarse comas para separarlas. Este campo también puede dejarse en blanco si fuera necesario.

Veamos otro ejemplo que utiliza lo anteriormente expuesto en combinación con la estructura condicional `if` para dibujar un árbol (o una flecha, según se mire) en pantalla tal y como se ilustra en la figura 3.1.

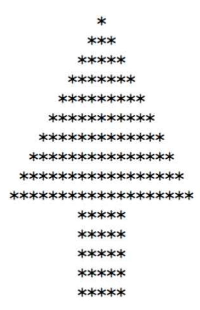
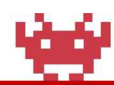


**Figura 3.1:** Árbol dibujado por el código del ejemplo 3.6.



**Ejemplo 3.6: El árbol**

Realiza un programa que dibuje un árbol con asteriscos por pantalla.

**Código**

```
#include <stdio.h>
int main()
{
    int i, j; // i varia con filas, j con columnas

    //Para cada fila...
    for(i=0;i<15;i++)
    {
        //Para las 10 primeras filas
        //se dibujan las  hojas
        if(i<10)
        {
            for(j=1;j<=20;j++)
            {
                if(j>=10-i&&j<=10+i)
                    printf("*");
                else
                    printf(" ");
            }
        }
        else  //para el resto el tronco
        {
            for(j=1;j<=20;j++)
            {
```

```
                if(j>=8&&j<=12)
                printf("*");
                else
                    printf(" ");
            }
        }

        printf("\n");
    }
}
```

El ejemplo 3.6 combina de forma anidada las estructuras presentadas en el capítulo. Se dibujan 15 líneas por pantalla (`i` varía de 0 a 14, ambos incluidos), cada una con 20 caracteres (`j` varía de 1 a 20, ambos incluidos). Lo que se dibuja en pantalla para las líneas comprendidas entre 0 y 9 viene determinado por la estructura condicional `if(i<10)`. En estas líneas se dibujan 20 caracteres por medio de **`for`**, que pueden ser un asterisco o un espacio en blanco, lo que depende del **`if`** anidado dentro de este for. Lo que sucede para las líneas de la 10 a la 14 viene determinado por el **`else`** del primer **`if`**. Nuevamente se utiliza un bucle **`for`** para imprimir 20 caracteres en pantalla, aunque esta vez la delimitación entre espacios en blanco y caracteres permanece constante.

## 3.5. Un juego iterativo

Como a programar solo se aprende programando, es hora de poner en práctica los conocimientos adquiridos. Para ello desarrollaremos un juego que utiliza las estructuras de control de flujo estudiadas y algunas de las instrucciones introducidas en el capítulo anterior. Se trata de un sencillo juego de adivinación, que nos servirá como calentamiento para lo que está por venir en el resto del capítulo. Una vez aprendáis a utilizar vectores y matrices en C programaremos juegos mucho más sofisticados. Ya lo veréis.



**Juego 3.1: Adivina**

Programa un juego de adivinación para dos jugadores en el que el primer jugador introduce un número entero y el segundo debe adivinarlo. El programa debe dar pistas al segundo jugador cada vez que se equivoque. De este modo, indicará si el número que ha introducido es mayor o menor que el que está buscando. El programa debe imprimir al finalizar el número de intentos del segundo jugador hasta adivinar el número.

**Código**

```
#include <stdio.h>
int main()
{
    int numJ1, numJ2, i, numintentos=0;

    printf("J1 intro num:"); // J1 introduce su numero
    scanf("%d", &numJ1);

    //Se "borra" la pantalla imprimiendo enter
    for(i=1;i<=100;i++)
    printf("\n");

    do   //J2 comienza a jugar
    {
        numintentos=numintentos+1;
        printf("\n J2 intro num:");
        scanf("%d", &numJ2);

        if(numJ2>numJ1)
        printf("\n Te pasaste.");

        if(numJ2<numJ1)
        printf("\n Te quedaste corto.");
    }
    while(numJ2!=numJ1);

    printf("\n Acertaste tras %d intentos.\n",
        numintentos);
}
```

El juego 3.1 pide un número al primer usuario de la manera estudiada en programas anteriores. A continuación borra la pantalla para evitar que el segundo jugador pueda ver el número introducido, lo que se consigue imprimiendo el carácter nueva línea `\n` 100 veces. Así, cuando el jugador 2 introduce su número ya no puede ver la información introducida por el jugador 1. Se utiliza un `do/while` para pedir al jugador 2 que introduzca su respuesta tantas veces como sea necesario hasta que acierte. Dado que la respuesta ha de introducirse al menos una vez, la elección de `do/while` resulta muy apropiada. Dentro del bucle se usa un `if` para dar pistas al jugador 2 en caso de que se equivoque. La condición de salida del bucle es la coincidencia entre los números introducidos por ambos jugadores. Cuando el bucle termina se muestra por pantalla el número de intentos hasta acertar el número.

## 3.6. Vectores y matrices en C

Explotar todo el potencial que ofrecen los bucles requiere ampliar nuestros conocimientos acerca de la forma en que se reservan variables en C. Para ello, se introducirá formalmente la definición de vectores y matrices en este lenguaje de programación.

Supongamos que necesitamos una zona en la memoria para almacenar todas las notas de los alumnos de una escuela o el nivel de gris que tiene cada uno de los píxeles de una foto en blanco y negro. Una posible solución sería declarar una variable nueva para cada elemento, es decir, una variable por alumno y una variable por píxel. Si la escuela tiene cien alumnos y la foto un millón de píxeles se necesitan... ¿cuántas variables? Ni que decir tiene que esta forma de proceder no es práctica. Para dar respuesta a esta necesidad C permite declarar vectores[1] y matrices, es decir, agrupaciones de datos dentro de la memoria.

**Recuerda 3.4: Declaración de vectores y matrices**

La declaración de matrices en C se efectúa con la sintaxis que se indica a continuación.

**Código**

```
tipo_de_dato   nombre_matriz[dim1][dim2];
```

La única diferencia con respecto a la declaración de una variable como las que hemos visto estriba en las parejas de corchetes que se añaden tras el identificador, donde cada pareja de corchetes añade una dimensión a la agrupación de datos que se va a realizar. Dentro de los corchetes se indica el número de elementos existentes en *esa* dimensión de la matriz. Dado que un vector es visto en C como una matriz de dimensión uno, basta con una pareja de corchetes para declararlo. De esta manera, si queremos declarar un vector de números enteros que tiene 3 componentes deberíamos escribir `int vector[3]`. Si lo que queremos es declarar una matriz $5 \times 5$ escribiríamos `int matriz[5][5]`. En este punto conviene destacar que pueden seguir añadiéndose corchetes si se desea reservar matrices con más de dos dimensiones, aunque este tipo de matrices no se verá dentro del libro.

La pregunta que podría surgir a continuación es de qué manera se le puede pedir al procesador que acceda al interior de un determinado elemento de una matriz. Una vez más, la notación de C es muy coherente. Acceder a la posición $(i, j)$ de `matriz` es tan sencillo como indicar `matriz[i][j]`. Análogamente, para acceder a la posición $i-$ésima de un vector se escribe `vector[i]`. Esta sencillez se ve ligeramente emborronada por la particular manera en que se numeran las filas y las columnas en el lenguaje C.

[1] Los vectores son también conocidos como arreglos o *arrays* en C.



**Recuerda 3.5: Límites para vectores y matrices en C**

El índice para referirse a la primera fila o columna de una matriz $M \times N$ en C es el 0. Como consecuencia, la última fila vendrá dada por $M - 1$ y la última columna por $N - 1$. Por ejemplo, dentro del siguiente código `mat[0][0]` y `v[0]` representan, respectivamente, el primer elemento de la matriz `mat` y del vector `v`. El último elemento de ambos vendrá dado por `mat[9][4]` y `v[8]`.

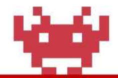
**Código**

```
int mat[10][5];
int v[8];
```

Respetar los límites anteriormente indicados es fundamental, ya que de no hacerlo se corre el riesgo de acceder a posiciones de memoria fuera del espacio reservado para la matriz o el vector, lo que puede ocasionar fallos e incluso el cierre inesperado del programa durante su ejecución.

Hay otra cuestión que podría pasar desapercibida y que no es trivial: ¿cómo se almacena una matriz que puede tener múltiples dimensiones en una estructura unidimensional como la memoria? Lo que nosotros sabemos de la calle memoria es que es un pasillo muy largo repleto de armarios. La cuestión admite discusión y varias soluciones posibles, pero a nosotros solo nos interesa la de C: linealizando la matriz por filas. Supongamos el caso más sencillo: una matriz bidimensional. En tal caso, la matriz se descompone en filas y cada una de estas filas es almacenada en memoria de forma independiente. Gráficamente esto puede verse como construir una serpentina a partir de una hoja de papel: se corta la hoja de papel, que es una estructura bidimensional, en finas tiras que pueden solaparse para construir una serpentina, que en este ejemplo hace las veces de estructura unidimensional. ¿Y si la matriz es tridimensional como un cubo de Rubik? En ese caso primero se divide la matriz en submatrices de dimension dos. Imaginad un cubo de plastilina que se cortara en *lonchas* bidimensionales. Cada una de estas lonchas es tan fina como las hojas de papel del ejemplo anterior, así que ya sabemos el segundo paso: cortar las lonchas en tiras y unir todo para poder formar una serpentina, la cual sí puede ser almacenada en la memoria. Si hubiera más dimensiones, el proceso se repite análogamente hasta reducirlo todo a una estructura unidimensional.

Veamos algunos ejemplos clásicos del uso de vectores y matrices en C.



**Ejemplo 3.7: Lectura e impresión de vectores**

Realiza un programa que lea un vector e imprima su contenido por pantalla.

**Código**

```
1  #include <stdio.h>
2  #define N 5
3
4  int main()
5  {
6      int v[N];
7      int i;
8
9      for(i=0; i<N; i++)
10     {
11         printf("Intro elem. %d del  vector: ", i);
12         scanf("%d", &v[i]);
13     }
14
15     printf("El vector es: ");
16     for(i=0; i<N; i++)
17         printf("%d ", v[i]);
18     printf("\n");
19 }
```

El ejemplo 3.7 utiliza sendos bucles `for` para leer y escribir el vector componente a componente gracias a la variable `i`, cuyo valor varía dentro del bucle de uno en uno entre 0 y `N-1`. (Nótese que la instrucción `i++` es una abreviatura de `i=i+1`. Análogamente, `i--` es equivalente a `i=i-1`.)

En este ejemplo se utiliza también la directiva del preprocesador `#define`, la cual establece para este programa la equivalencia entre `N` y `5`. Esta directiva sustituye cada aparición de `N` por 5 justo antes de compilar el código. Gracias a ello, se evita tener que escribir los límites del vector numéricamente tanto en la declaración como en la condición de los `for`. Asimismo, si se quiere modificar el código para que funcione correctamente con vectores de dimensión 10 bastará con escribir únicamente `#define N 10` en la línea 2, ahorrándonos así el trabajo de cambiar manualmente dentro del código cualquier referencia existente al tamaño del vector.



**Ejemplo 3.8: Ordenación de los elementos de un vector**

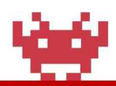
Realiza un programa que lea un vector de números y ordene sus componentes de menor a mayor. Para ello, recorrerá el vector tantas veces como sea necesario, intercambiando los elementos desordenados en posiciones consecutivas.

**Código**

```
1  #include <stdio.h>
2  #define N 5
3  int main()
4  {
5      int v[N];
6      int i, aux, flag=0;
7
8      for(i=0; i<N; i++)
9      {
10         printf("Intro elem. %d del  vector: ", i);
11         scanf("%d", &v[i]);
12     }
13
14     do
15     {
16         flag=0;
17         for(i=0;i<N-1;i++)
18         {
19             if(v[i+1]<v[i])
20             {
21                 aux=v[i+1];
22                 v[i+1]=v[i];
23                 v[i]=aux;
24                 flag=1;
25             }
26         }
27     }
28     while(flag==1);
29
30     printf("El vector es: ");
31     for(i=0; i<N; i++)
32         printf("%d ", v[i]);
33     printf("\n");
34 }
```

El ejemplo 3.8 es una variación del ejemplo 3.7 en el que se añade un trozo de código adicional (líneas 14 a 26), que se encarga de ordenar las componentes del vector. La estrategia que se sigue para ello es conocida como método de la burbuja. La idea consiste en recorrer los elementos del vector del primero al penúltimo comparando a cada uno con el que le sigue (líneas 17 a 26). Cada vez que se detecta un par de elementos consecutivos con un orden que no se corresponde con el deseado (línea 19) se realiza un intercambio de posiciones con la ayuda de la variable **aux** (líneas 21 a 23).

Este proceso se repite con **do/while** tantas veces como sea necesario. En concreto, la condición para dejar de iterar es que la variable **flag** adopte el valor 0, algo que sucede solamente si se consigue recorrer el vector de principio a fin sin realizar ninguna permutación entre componentes consecutivas.



**Ejemplo 3.9: El palíndromo**

Realiza un programa que detecte si una palabra introducida por teclado es un palíndromo, esto es, una palabra que se lee igual de izquierda a derecha y de derecha a izquierda.

**Código**

```
#include <stdio.h>

int main()
{
    int i, lon, flag=1; //flag =1 -> es un palindromo
    char cad[100];

    printf("Intro cadena:");
    gets(cad);

    i=0;
    while(cad[i]!='\0')
    {
        i=i+1;
    }

    lon=i-1;

    for(i=0; i<lon/2; i=i+1)
    {
        if(cad[i]!=cad[lon-i])
        flag=0;  //no palind.
    }

    if(flag==1)
    printf("Es un palindromo\n");
    else
        printf("No es un palindromo\n");
}
```

En el ejemplo 3.9 se utiliza un vector de caracteres llamado `cad` para almacenar el texto que se introduce por teclado y una variable auxiliar llamada `flag` a la que daremos el valor 1 si `cad` es un palíndromo y 0 en caso contrario. Como a priori no sabemos si la palabra que introducirá el usuario será un palíndromo, le daremos el beneficio de la duda y asumiremos que sí lo es. Por tanto, nuestro código deberá verificar si esta asunción es correcta.

Cuando el usuario introduce su palabra y pulsa enter, la información es almacenada en memoria gracias a la función `gets` (línea 9). En general, las funciones que recogen texto desde el teclado añaden a lo introducido por el usuario el carácter '`\0`' a modo de terminador. Nuestro código explota esto para detectar con un bucle `while` el final de la información introducida (líneas 12 a 15). Al salir del bucle, se le da a la variable `lon` el valor necesario para que `cad[lon]` se corresponda con el último carácter del texto del usuario (línea 17). De este modo, se procede a comparar sucesivamente el primer carácter de la palabra con el último, el segundo con el penúltimo, etc., lo que se hace dentro del bucle `for` de las líneas 19 a 23. Si en alguna de estas comparaciones se detectan caracteres distintos, la palabra no es un palíndromo, por lo que se cambia a 0 el valor de la variable `flag`. En cambio, si el bucle termina y `flag` aún vale 1 sabremos con certeza que nuestra suposición inicial era correcta.

Nótese el papel esencial del terminador '`\0`' en el ejemplo 3.9, pues permite distinguir qué parte del vector de caracteres pertenece a la información introducida por el usuario y qué parte no. A los vectores de caracteres que cuentan con el terminador '`\0`' se les llama también cadenas de caracteres, ya que por su importancia en C merecen nombre propio.



**Ejemplo 3.10: Lectura e impresión de una matriz**

Realiza un programa que lea una matriz de números enteros y la muestre por pantalla.

**Código**

```
#include <stdio.h>
#define M 3
#define N 4

int main()
{
    int matriz[M][N];
    int i, j;

    //leer matriz
    for (i=0;i<M;i++)
```

```
12      {
13          for (j=0;j<N;j++)
14          {
15              printf("\n Elemento [%d][%d]: ",i,j);
16              scanf("%i",&matriz[i][j]);
17          }
18      }
19
20      //imprimir matriz
21      printf("La matriz es:\n");
22      for (i=0;i<M;i++)
23      {
24          for(j=0;j<N;j++)
25              printf ("%d ",matriz[i][j]);
26
27          printf ("\n");
28      }
29  }
```

El ejemplo 3.10 es una extensión del ejemplo 3.7 al caso matricial. Dado que una matriz es bidimensional, se necesita un bucle `for` anidado dentro de otro para recorrer todos sus elementos tanto para leer como para imprimir. Por ejemplo, en la lectura de los elementos de la matriz el bucle externo (líneas 11 a 18) utiliza la variable auxiliar `i` para recorrer cada fila. Dentro de este bucle se ejecuta otro (líneas 13 a 17) que utiliza la variable `j` para recorrer todas las columnas contenidas en dicha fila y pedir al usuario que introduzca el valor correspondiente mediante `scanf`. En la impresión de la matriz por pantalla (líneas 21 a 28) se sigue la misma estrategia.



**Ejemplo 3.11: Matriz traspuesta**

Realiza un programa que lea una matriz de números enteros, la muestre por pantalla y que calcule y muestre su traspuesta, es decir, la matriz resultante al cambiar filas por columnas

**Código**

```
#include <stdio.h>
#define M 3
#define N 4

int main()
{
```

```
    int matriz[M][N];
    int mt[N][M];
    int i, j;

    //leer matriz
    for (i=0;i<M;i++)
    {
        for (j=0;j<N;j++)
        {
            printf("\n Elemento [%d][%d]: ",i,j);
            scanf("%i",&matriz[i][j]);
        }
    }

    //trasponer matriz
    for (i=0;i<M;i++)
        for (j=0;j<N;j++)
            mt[j][i]=matriz[i][j];

    //imprimir matriz original
    printf("La matriz traspuesta es:\n");
    for (i=0;i<M;i++)
    {
        for(j=0;j<N;j++)
            printf ("%d ",matriz[i][j]);
         printf ("\n");
     }

     //imprimir matriz traspuesta
     printf("La matriz es:\n");
     for (i=0;i<N;i++)
     {
          for(j=0;j<M;j++)
              printf ("%d ",mt[i][j]);
          printf ("\n");
     }
}
```

El ejemplo 3.11 es una ampliación del ejemplo 3.10 en el que se reserva espacio adicional para la matriz `mt`, que es donde se almacenará la traspuesta. Una vez introducida la matriz `matriz` se utilizan dos bucles `for` para asignar al elemento (`j`,`i`) de `mt` el valor almacenado en el elemento (`i`,`j`) de `matriz`. Finalmente se imprimen ambas matrices por pantalla.

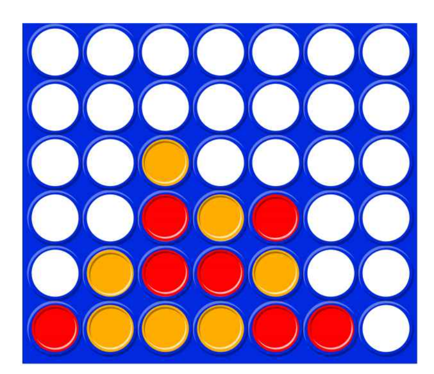


**Figura 3.2:** Tablero del 4 en raya (ilustración por cortesía de François Haffner).

## 3.7. Juegos con vectores y matrices

Los ejemplos anteriores habrán ayudado a consolidar vuestro manejo de bucles, vectores y matrices. Es el momento de buscar aplicaciones más lúdicas de los conocimientos adquiridos.

Comenzaremos con una versión en C del juego de mesa conocido como 4 en raya, que se ilustra en la figura 3.2.



**Juego 3.2: Conecta 4**

Conecta 4, también conocido como 4 en raya, es un popular juego en el que dos jugadores se alternan para introducir fichas de colores en un tablero vertical con el objeto de alinear cuatro del mismo color de forma consecutiva, ya sea en horizontal, vertical o diagonal. Programa una versión en C de este juego.

**Código**

```
#include <stdio.h>
#define FILAS 8
#define COLUMNAS 8

int main()
{
    char tablero[FILAS][COLUMNAS];
    char simbolo[2]={'+','*'};
    int i, j; //var aux para matrices
    int turno; //para saber quien va
```

```
    int col; // columna escogida para ficha
    int  numjugadas=0; // contador num jugadas
    int gameover=0; //vale 1 al acabar juego
    int ok; // indica si jugada es correcta

    for(i=0;i<FILAS;i++)
    for(j=0;j<COLUMNAS;j++)
    tablero[i][j]='O';

    while(gameover==0)
    {
        //determinar a quien le toca
        turno=numjugadas%2+1;
        numjugadas++;

        //imprimir tablero
        printf("\n\n\nColumnas a elegir:\n");
        for(j=0;j<COLUMNAS;j++)
        printf("%d ",j);
        printf("\n");

        for(i=0;i<FILAS;i++)
        {
            for(j=0;j<COLUMNAS;j++)
            printf("%c ",tablero[i][j]);

            printf("\n");
        }

        //turno jugador
        do
        {
            ok=0;
            printf("\nLe toca a jugador %d. Elija columna
                : ", turno);
            scanf("%d",&col);

            if(col>=0&&col<COLUMNAS)
            {
                //encontrar fila en la que insertar
                for(i=FILAS-1;tablero[i][col]!='O'&&i>=0;i
                    --);

                if(i>-1)
```

```
                {
                    ok=1;
                    tablero[i][col]=simbolo[turno-1];
                }

            }
        }
        while(!ok);

        //Comprobar si hay ganador
        //Busqueda de 4 en raya en horizontal
        for(i=0;i<FILAS;i++)
        for(j=0;j<COLUMNAS-3;j++)
        if (tablero[i][j]!='O' && tablero[i][j]==tablero
            [i][j+1] && tablero[i][j]==tablero[i][j+2]
            && tablero[i][j]==tablero[i][j+3])
            gameover=1;

        //Busqueda de 4 en raya en vertical
        for(i=0;i<FILAS-3;i++)
        for(j=0;j<COLUMNAS;j++)
        if (tablero[i][j]!='O'&& tablero[i][j]==tablero[
            i+1][j] && tablero[i][j]==tablero[i+2][j] &&
             tablero[i][j]==tablero[i+3][j])
            gameover=1;

        //Busqueda diagonal abajo-derecha
        for(i=0;i<FILAS-3;i++)
        for(j=0;j<COLUMNAS-3;j++)
        if (tablero[i][j]!='O' && tablero[i][j]==tablero
            [i+1][j+1] && tablero[i][j]==tablero[i+2][j
            +2] && tablero[i][j]==tablero[i+3][j+3])
            gameover=1;

        //Busqueda diagonal abajo-izquierda
        for(i=0;i<FILAS-3;i++)
        for(j=3;j<COLUMNAS;j++)
        if (tablero[i][j]!='O'&& tablero[i][j]==tablero[
            i+1][j-1] && tablero[i][j]==tablero[i+2][j
            -2] && tablero[i][j]==tablero[i+3][j-3])
            gameover=1;

    }

```

```
89     printf("\n\n\nHa ganado el jugador %d tras %d jugadas!!!",turno,numjugadas);
90
91     //impresion final tablero
92     printf("\n\n\n");
93     for(i=0;i<FILAS;i++)
94     {
95         for(j=0;j<COLUMNAS;j++)
96         printf("%c ",tablero[i][j]);
97
98         printf("\n");
99     }
100 }
```

Como vemos, el código del juego 3.2 es notablemente largo en comparación con todos los programas realizados hasta el momento. Tras la declaración de las variables que serán utilizadas en el programa, se incluyen unas líneas para inicializar todos los elementos de la matriz `tablero` con el carácter 'O'. Nótese que los símbolos empleados para representar las fichas de los jugadores se almacenan en el vector `simbolo` (las fichas del jugador 1 se representan con '+' y las del segundo con '*').

El grueso del código se encuentra entre las líneas 20 y 87, donde se utiliza un bucle `while` para realizar las siguientes tareas:

- Cálculo del jugador al que le toca (líneas 22 a 24): se acumula el número de jugadas realizadas hasta el momento en la variable `numjugadas` y se utiliza el resto de dividir su valor entre dos como base para saber si le toca al primer jugador (`turno`=1) o al segundo (`turno`=2).
- Impresión del tablero (líneas 26 a 38): se imprime el número de cada columna por pantalla y a continuación la matriz `tablero`.
- Elección de columna por parte del jugador al que le toca (líneas 41 a 60): dentro de un bucle `do/while` se pregunta al jugador por la columna elegida. Si la elección es correcta (columna dentro de los límites de la matriz (línea 47) y tiene espacio disponible (líneas 50 a 52), se introduce la ficha con el símbolo correspondiente y se hace `ok`=1, lo que permite la ruptura del bucle. Destacan en este trozo del código dos aspectos:
    - El bucle `for` de la línea 50 repite una instrucción vacía ya que lleva punto y coma al final. Esto se utiliza para empaquetar en una sola línea todo lo necesario para encontrar el primer elemento de la columna escogida en el que se puede introducir una ficha. Como no hay ninguna instrucción que repetir se pone directamente ';' tras `for`. Nótese que el bucle empieza por la última fila de la matriz y retrocede mientras que no se encuentre un hueco disponible e `i>=0`. A la salida, se comprueba el valor de `i` en el `if`

para determinar cuál de las dos condiciones propició la ruptura del bucle, de modo que solo se inserta la ficha si se encontró un hueco disponible.

- La condición de ruptura de `do/while` es `!ok`, que es equivalente a `ok==0`. En efecto, el operador negación '`!`' conmuta la interpretación lógica de lo que le sucede. Así, si `ok` es igual a 0, `!ok` será interpretado como algo diferente de 0, lo que desde el punto lógico se interpreta como verdadero. Si `ok` vale algo diferente de cero, `!ok` pasa a ser cero, algo que desde el punto de vista lógico se interpreta como falso. Por tanto, al hacer `ok`=1, la condición `!ok` rompe el bucle. Nótese que podría haberse utilizado también este recurso en la línea 20 con la condición del `while`.

- Búsqueda de 4 en raya (líneas 62 a 85): el programa comprueba si la ficha introducida lleva a la finalización del programa. Ello requiere recorrer la matriz y comprobar si la ficha de la posición (`i`,`j`) es igual a las que la suceden en horizontal (líneas 62 a 67), vertical (líneas 69 a 73) y diagonal (líneas 75 a 85). Cada uno de los bucles `for` empleados ajusta el rango de variación de las variables `i` y `j` para evitar que durante la comprobación se acceda a una casilla fuera de los límites definidos para la matriz. Si se encuentran 4 en raya, la variable `gameover` pasa a valer 1, lo que desencadena el fin del juego. Nótese que los `for` e `if` aquí utilizados afectan respectivamente a la instrucción que los sucede, por lo que pueden omitirse las llaves, lo que aligera el número de líneas del código.

Desde la línea 89 hasta el final se indica el jugador vencedor y se imprime la disposición final del tablero.

En la figura 3.3 se muestra una captura de pantalla del aspecto del programa en ejecución.

```
Columnas a elegir:
0 1 2 3 4 5 6 7
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 * 0 0 0 0
0 * + + + * 0 0

Le toca a jugador 1. Elija columna:
```

**Figura 3.3:** Tablero del 4 en raya de nuestra versión en C.




**Figura 3.4:** Pantalla de inicio de *The Secret of Monkey Island*.

El siguiente juego que desarrollaremos está inspirado en una aventura gráfica llamada *The Secret of Monkey Island*, lanzada al mercado en 1990 por LucasFilm Games (sí, una empresa relacionada con George Lucas, el de *La Guerra de las Galaxias*). En la figura 3.4 podéis ver la pantalla de inicio del juego. En esta aventura encarnamos a Guybrush Threepwood, un muchacho que quiere ser pirata y que se ve envuelto en una de las tramas más delirantes y originales de la historia de los videojuegos. Pese a que los años le han pasado factura a los gráficos, la diversión que ofrece su soberbio guión permanece intacta. Tanto es así, que *The Secret of Monkey Island* y todas sus secuelas han sido versionados en multitud de ocasiones, de manera que con toda probabilidad encontraréis alguna versión disponible para PC, consola o teléfono móvil.

Como último juego del capítulo presentamos un programa que recrea las batallas de piratas de *The Secret of Monkey Island*. La mecánica de las mismas es muy peculiar, ya que no se combate a golpes sino a insulto limpio. Una batalla de gallos en estilo pirata.



**Juego 3.3: Batallas en la Isla del Mono**

Programa un juego en el que ambos jugadores tengan a su disposición una serie de frases de ataque y de respuesta. El jugador en posesión del turno insultará a su rival con una de las frases de ataque y recibirá una respuesta por parte de su contrincante. Si la respuesta al insulto es buena, el rival recupera el turno para sí mismo. En caso contrario, el rival pierde un punto de vida y el turno sigue en posesión del jugador atacante. Como novedad, se desarrollará el juego de manera que uno de los jugadores sea el propio ordenador.

**Código**

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define NUMATAQUES 10
#define NUMRESPUESTAS 5
#define PTOSVIDA 3
#define CPU 0
#define PLAYER 1

int main()
{
    // lista de opciones de ataque
    char ataques[NUMATAQUES][100]={"Luchas como un
        ganadero!", "Mi lengua es más hábil que
        cualquier espada.", "Ordeñaré hasta la última
        gota de sangre de tu cuerpo!", "Ya no hay
        técnicas que te puedan salvar.","No hay
        palabras para describir lo asqueroso que eres!"
        , "Llevaras mi espada como si fueras un pincho
        moruno!", "He hablado con simios más educados
        que tu!", "Ahora entiendo lo que significan
        basura y estupidez.", "Obtuve esta cicatriz en
        mi cara en una lucha a muerte!", "Acabe en mi
        última pelea con las manos llenas de sangre."};

    // lista de opciones de defensa
    char respuestas[NUMRESPUESTAS][100]={"Primero
        deberías dejar de usarla como un plumero.","Que
         apropiado, tu peleas como una vaca.", "Si que
        las hay, solo que nunca las has aprendido.", "
        Me alegra que asistieras a tu reunión familiar
        diaria.", "Espero que ya hayas aprendido a no
        tocarte la nariz."};

    // vector con indice de resp correcta a cada ataque
    int respcorrecta[NUMATAQUES]={1,0,1,2,2,0,3,3,4,4};

    // variables auxiliares
    int i, turno, iataq, iresp, gameover=0;
    int vidapirata=PTOSVIDA, vidajugador=PTOSVIDA;

```

```
    // inicializacion de la generacion de num aleatorios
    srand(time(NULL));
    turno=rand()%2;    //decide quien empieza

    // Compienza el juego...
    printf("\n Viajas por un camino y te encuentras con un pirata!");

    while(gameover==0)  //bucle del juego
    {
        if(turno==CPU)   //le toca al pirata
        {
            iataq=rand()%NUMATAQUES; //seleccion aleatoria de ataque
            printf("\n El pirata dice:");
            printf("\n %s", ataques[iataq]);

            printf("\n Dile algo:\n");  //usuario responde
            for(i=0; i<NUMRESPUESTAS; i++)
                printf("%d - %s \n", i, respuestas[i]);
            printf("Escoge una opcion: ");
            scanf("%d",&iresp);

            //si resp es correcta -> le toca a jugador
            if(respcorrecta[iataq]==iresp)
            {
                printf("\nEl pirata dice: ahhh!");
                turno=PLAYER;
            }
            else  // si no, sigue pirata y jugador es herido
            {
                printf("\n El pirata dice: ja, ja, toma!!");
                vidajugador--;
            }
        }
        else //turno del jugador
        {
            printf("\n Dile algo:\n");  //eleccion de ataque
            for(i=0; i<NUMATAQUES; i++)
```

```
                printf("%d - %s \n", i, ataques[i]);
            printf("Escoge una opcion: ");
            scanf("%d", &iataq);

            //pirata responde al azar
            iresp=rand()%NUMRESPUESTAS; //seleccion aleatoria de ataque
            printf("\n El pirata responde:");
            printf("\n %s", respuestas[iresp]);

            //si resp es correcta, le toca a pirata
            if(respcorrecta[iataq]==iresp)
            {
                printf("\nEl pirata dice: esta me la sabia!");
                turno=CPU;
            }
            else   // si no, jugador hiere a pirata
            {
                printf("\n El pirata dice: ah, me has dado!!");
                vidapirata--;
            }
        }

        //Se comprueba si algun jugador perdio
        if(vidapirata==0||vidajugador==0)
            gameover=1;
    }

    // Gana el jugador que mantiene el turno
    // Mensajes de fin de juego
    if(turno==CPU)
        printf("\n El pirata dice: has perdido, grumete.\n");
    else
        printf("\n El pirata dice: Me rindo! Suerte en tu viaje.\n");
}
```

El juego 3.3 se diferencia de todos los anteriores porque es el primero en el que la máquina toma un papel protagonista al convertirse en uno de los jugadores. Para ello, incluiremos en nuestro código dos librerías nuevas que nos proporcionarán funcionalidades extra de gran interés en este sentido:

- `stdlib.h`: se incluye para utilizar la instrucción `rand()` en la líneas 28, 37 y 68, la cual genera un número entero positivo aleatorio. Al calcular el resto de la división del valor devuelto por `rand()` entre un número dado *num*, se obtiene un valor aleatorio entre 0 y $num - 1$. De esta manera se consigue elegir al azar quién comienza primero el juego (línea 28). Más adelante, se utiliza para que la máquina elija sus frases de ataque (línea 37) y de respuesta (línea 68) aleatoriamente.

  La función `rand()` necesita un poco de ayuda para garantizar la aleatoriedad de los números que devuelve. En concreto, pide que se le proporcione un número aleatorio como *semilla*. Curioso, ¿verdad? La función que proporciona números aleatorios necesita un numero aleatorio para funcionar bien. Para una misma semilla dada, `rand()` devuelve siempre la misma secuencia de números cada vez que es invocada, así que en sentido estricto `rand()` proporciona números aleatorios a medias (pseudoaleatorios en la jerga informática). La semilla que `rand()` utiliza se proporciona entre paréntesis a través de la instrucción `srand()` (línea 27).

  Para asegurar una cierta aleatoriedad en la semilla lo que se le pasa a la función es la hora que el sistema alberga en su interior, que se almacena con tanta precisión (con milisegundos y todo) que a efectos prácticos puede considerarse un número aleatorio.

- `time.h`: esta librería se incluye para utilizar la instrucción `time(NULL)` en la línea 27. Básicamente esta instrucción devuelve la hora que el ordenador tiene internamente, que como ya dijimos se utiliza como semilla para la generación de números aleatorios con `rand()`.

Tras la inclusión de estas librerías, el código continúa con la definición de una serie de constantes numéricas mediante la directiva `#define`, que, como siempre, simplifican la escritura y la interpretación del resto del código. En la medida de lo posible hay que evitar utilizar números dentro del código, debido a que un número aislado es difícil de interpretar. Conviene recordar que el código debe escribirse siempre pensando en que otras personas pueden leerlo.

Una vez dentro de `main`, se comienza declarando e inicializando las matrices de caracteres `ataques` y `respuestas` y el vector de enteros `respcorrecta` (líneas 14 a 20). Como puede verse, C permite proporcionar un valor conocido de antemano de la forma que se indica en el código. En el caso de las matrices de caracteres, cada una de las frases se almacena dentro de una fila de la matriz, cuyo número de columnas debe ser lo suficientemente grande como para almacenar a la frase más larga de todas. De este modo, para imprimir en pantalla una de estas frases basta indicar la fila correspondiente de la matriz (líneas 43 y 63). El vector `respcorrecta` tiene tantos elementos como frases de ataque y almacena en cada uno de ellos el índice de la fila con la respuesta correcta en la matriz `respuesta`. Por ejemplo, el elemento 0 de este vector vale 1, lo que indica que la respuesta correcta a la frase en la fila 0 de `ataques` (`Luchas como un ganadero!`) está en la fila 1 de `respuestas` (`Que apropiado, tú peleas como una vaca.`). Este vector proporciona la clave pa-

ra comprobar si la respuesta dada por el usuario (línea 46) o la CPU (línea 73) es o no correcta.

La estrategia de la máquina como jugador en este juego es muy sencilla: básicamente no hay estrategia. Todo se decide al azar gracias al uso de las librerías anteriormente indicadas. Cuando le toca a la máquina (líneas 35 a 57), se elige una frase de ataque al azar (líneas 37 a 39) y se espera a la respuesta del usuario (líneas 41 a 45). Si la respuesta es correcta conforme a lo indicado en el vector `respcorrecta`, el pirata gime de dolor y pierde el turno (líneas 47 a 52). Por el contrario, si el usuario yerra en la respuesta, pierde un punto de vida y el pirata ríe (líneas 53 a 57). Cuando le toca al jugador (líneas 59 a 83) el proceso es justamente el inverso: el usuario elige frase de ataque (líneas 61 a 65) y recibe una respuesta aleatoria de la CPU (líneas 67 a 70). Si la respuesta de la máquina es correcta (líneas 73 a 77), el usuario pierde el turno mientras el pirata se jacta de su acierto. Si no, el pirata pierde vida e indica que le han dado (líneas 78 a 82).

El juego termina cuando alguno de los dos jugadores pierde todos sus puntos de vida, algo que se comprueba en la línea 86. Cuando esto sucede, la variable `gameover` pasa a valer 1, lo que rompe el bucle del juego (`while` de la línea 33). Tras ello, el pirata dice algo en función del resultado del juego (líneas 92 a 95), que se sabe gracias a la variable `turno`. En concreto, el jugador que tenía el turno al romper el bucle del juego es el ganador, ya que solo se puede dañar al rival cuando se tiene el turno.

Con esto concluimos el último programa del capítulo, que nos ha dado las herramientas necesarias para introducir aleatoriedad en nuestros juegos. A partir de ahora ni siquiera nosotros sabremos lo que sucederá en cada partida. Y esto es una excelente noticia, ya que la diversión se alimenta de lo inesperado.

## 3.8. Ejercicios propuestos

Finalmente os proponemos los siguientes retos de programación para que practiquéis lo aprendido. A ver cuántos sois capaces de superar. Tampoco dejéis de pensar en vuestros propios juegos. Nada será tan eficaz para el aprendizaje como vuestros propios retos.



**Ejercicio 3.1: Adivina contra la máquina**

Modifica el juego 3.1 para que el PC juegue como primer jugador. Por tanto, será la máquina la que genere un número aleatorio que un jugador humano tendrá que adivinar en el menor número de intentos posibles. Para simplificar las cosas, el programa comunicará al jugador al inicio los valores mínimo y máximo del número a adivinar.



**Ejercicio 3.2: Adivina para 2 jugadores**

El juego 3.1 deja al primer jugador a la espera de que el segundo adivine el número que ha introducido. Modifica el juego para que ambos jugadores introduzcan un número al inicio y se embarquen en una competición para ver quién adivina antes el número del otro. ¡El que pierda invita a la merienda!



**Ejercicio 3.3: Conecta 4 2.0**

La versión que hemos creado de Conecta 4 (juego 3.2) está diseñada bajo la premisa de que algún jugador ganará antes de llenar el tablero de fichas. ¿Qué sucede con el código actual cuando se da esta condición? Modifica el código para que cuando esto suceda se termine el programa y se indique por pantalla que el juego ha acabado en empate.



**Ejercicio 3.4: Batallas en la Isla del Mono mejorado**

¿Qué sucedería si en nuestra versión de *Monkey Island* (juego 3.3) el jugador introduce un número de ataque o respuesta que no está en la lista? Modifica el programa para contemplar este caso y así evitar problemas.



**Ejercicio 3.5: Juegos de combates por turnos**

El juego 3.3 introduce las bases para programar juegos de combates por turnos. Algunos videojuegos tan populares como *Final Fantasy* o *Pokemon* están basados en esta mecánica de combate. Anímate y recrea en C una batalla entre tus personajes favoritos. Si no se te ocurre nada siempre puedes programar una versión para PC de *Piedra, Papel o Tijera*, pero no será tan divertido...

# Capítulo 4

# Funciones

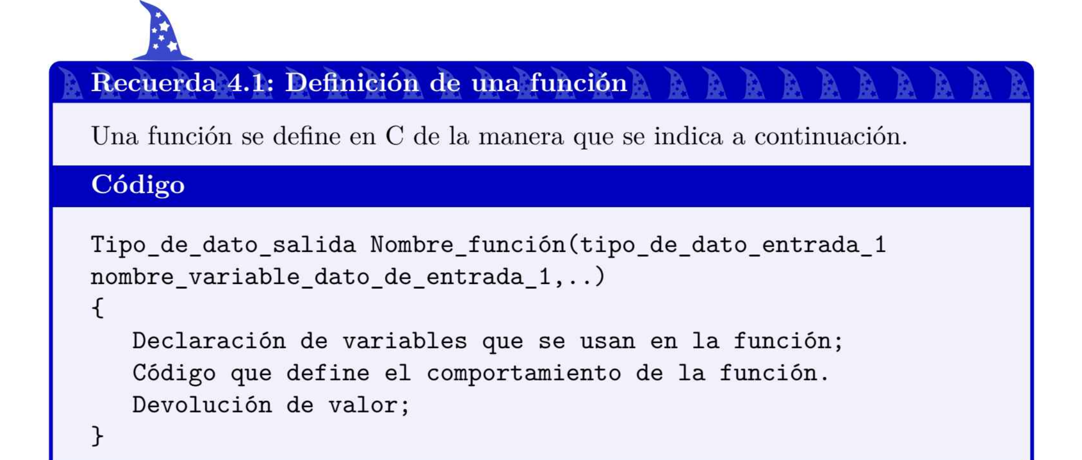
Las funciones son uno de los ingredientes más característicos de la programación en C. Tanto es así, que todo programa en C posee al menos una función principal, llamada *main*, que tiene que ser definida para que el programa tenga sentido. Pero, ¿qué es una función? Una función no es más que un conjunto de instrucciones en C que realizan una tarea común. Al separar un programa en funciones se evita que todo el código se concentre en la función principal, lo que simplifica su diseño y mantenimiento y reduce la probabilidad de cometer fallos de programación.

Por ejemplo, supongamos que nos contratan para organizar el trabajo dentro de un restaurante. Si es muy pequeño, una sola persona puede encargarse de todo el trabajo: atender a los clientes, cocinar, cobrar, y realizar la contabilidad; esa sería su *función* dentro del restaurante, hacer un poco de todo de todo. A medida que el restaurante crece en tamaño es difícil que todo lo haga la misma persona. Es mucho más razonable que haya una persona encargada de la contabilidad, otra de la cocina y así con cada una de las tareas propias del negocio. Es decir, estamos dividiendo el trabajo del restaurante en *funciones* que asignamos al personal que trabaja en el mismo. En programación se procede exactamente de la misma manera: los programas se subdividen en funciones que desarrollan una carga razonable de trabajo cada una.

En este capítulo aprenderemos a “encapsular” fragmentos de código con un propósito bien definido dentro de funciones, lo que nos permitirá crear instrucciones personalizadas dentro de nuestros programas. Se presentan también los conceptos relacionados con el intercambio de información entre la función y el resto del programa. Una vez dominado el contenido del capítulo, nuestras posibilidades como desarrolladores se multiplicarán, pues sabremos cómo usar las funciones realizadas por otros programadores (disponibles en Internet o en librerías como `stdio.h`) y podremos crear las nuestras propias en caso de que sea necesario.

## 4.1. Definición: anatomía de una función

Las funciones tienen su propia receta de elaboración. Aprendamos cómo se construye una.



**Recuerda 4.1: Definición de una función**

Una función se define en C de la manera que se indica a continuación.

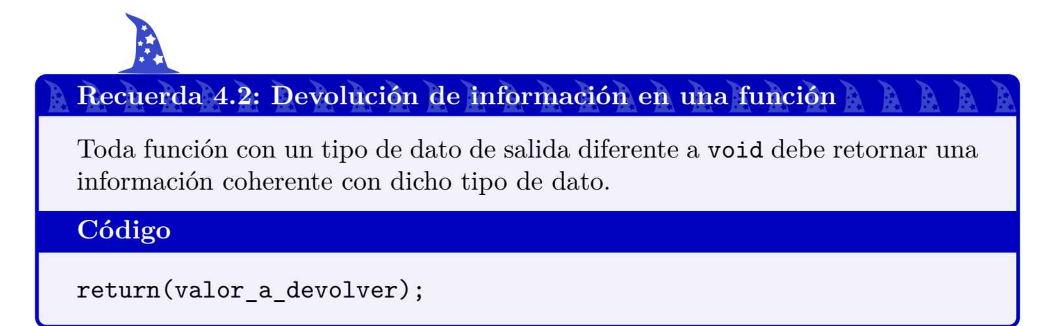
**Código**

```
Tipo_de_dato_salida Nombre_función(tipo_de_dato_entrada_1
nombre_variable_dato_de_entrada_1,..)
{
   Declaración de variables que se usan en la función;
   Código que define el comportamiento de la función.
   Devolución de valor;
}
```

A simple vista la sintaxis es un poco rebuscada, aunque hay poco que podamos hacer al respecto. La definición comienza con una primera línea que coloca como elemento central el nombre de la función, la palabra mágica que hay que utilizar para su invocación. A su derecha, entre paréntesis, aparecen una lista de argumentos de entrada, variables internas que la función utiliza para recibir información del exterior. Como veremos más adelante, la invocación de la función debe venir acompañada de los valores iniciales que tomarán estas variables. Por último, a la izquierda, se proporciona únicamente el tipo de dato que la función enviará al exterior. Cuando la función no devuelve nada, se indica `void` como tipo de dato, que podría traducirse en español como *vacío*. De alguna manera, esta primera línea describe cómo se debe realizar la interacción con la función, algo equivalente a saber utilizar una máquina de refrescos. Uno puede desconocer su funcionamiento interno, pero sabe que a la máquina se le proporcionan diferentes tipos de monedas y a cambio devuelve latas de refresco.

Tras la primera línea se escribe entre llaves la implementación de la función en C propiamente dicha, que se desarrolla de forma similar a los programas que hemos visto. No en vano hemos desarrollado siempre en todos nuestros programas una función llamada `main`. Por tanto, conocemos lo que tenemos que hacer: un programa con un propósito muy concreto. Así, las primeras líneas de la función se encargan de la declaración de las variables que vayamos a utilizar. Por supuesto, no debemos olvidar que todos los argumentos de entrada son también variables de la función que reciben su valor inicial en el momento de la invocación. Estas variables de entrada ya están declaradas y es importante abstenerse de re-declararlas en el interior de la función. Hacerlo solo generará problemas durante la compilación y ejecución del código. Recordadlo bien porque es un error habitual entre los estudiantes de programación.

Una vez se han declarado las variables (si es que se necesita alguna), lo siguiente es escribir las instrucciones que llevan a cabo la tarea encomendada a la función. Finalmente, si la función devuelve un tipo de dato diferente de `void` debe retornarse un valor mediante la instrucción `return` tal y como indica el siguiente cuadro.



**Recuerda 4.2: Devolución de información en una función**

Toda función con un tipo de dato de salida diferente a `void` debe retornar una información coherente con dicho tipo de dato.

**Código**

```
return(valor_a_devolver);
```

En la instrucción anterior, el parámetro de salida `valor_a_devolver` ha de tener el mismo tipo de dato especificado como salida de la función. Puede tratarse de una constante (un número, por ejemplo) o del nombre de una variable, en cuyo caso se devuelve el valor contenido en la misma. Una vez ejecutada la instrucción `return`, la función desaparece de memoria y el flujo del programa vuelve al punto en que se realizó su invocación.

## 4.2. Paso de parámetros

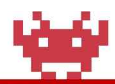
Como decíamos, una función puede verse como una máquina que puede recibir información y devolverla. Toda la información enviada durante la invocación de la función es transmitida a través de las variables de entrada, que como hemos visto son aquellas que aparecen entre paréntesis justo después del nombre de la función. Como el valor inicial de estas variables es recibido durante la invocación, es esencial que los datos enviados coincidan en tipo con los que se hayan declarado para los argumentos de entrada.

Según la naturaleza de las variables empleadas para esta transmisión de información se distinguen en C dos casos posibles, conocidos como paso por valor y paso por referencia. Veamos en qué consiste cada uno de ellos.

### 4.2.1. Paso por valor: trabajando con copias

Se dice que se trabaja por valor cuando la función recibe una copia de la información remitida desde el exterior, algo que sucede siempre que no se trabaje con vectores o matrices. De este modo, las variables definidas como receptoras de la información de entrada se inicializan con el valor transmitido. Estas variables son locales a la función y, por tanto, cualquier cambio que experimenten no afectará en modo alguno a

ninguna otra parte del programa. Asimismo, estas variables desaparecen de la memoria cuando la función acaba. De la misma manera, el valor devuelto con **return** es una copia de la información contenida dentro de la función. Por este motivo se trata también de una transmisión de información por valor. Veamos un ejemplo.



**Ejemplo 4.1: Función para multiplicar**

Realiza un programa que pida al usuario dos números por teclado y los multiplique utilizando una función.

**Código**

```
#include <stdio.h>

int producto(int a, int b)
{
    int c;
    c=a*b;
    return (c);
}

int main ()
{
    int numero1;
    int numero2;
    int resultado;

    printf("Introduzca el primer numero: ");
    scanf("%d",&numero1);
    printf("Introduzca el segundo numero: ");
    scanf("%d",&numero2);
    resultado=producto(numero1, numero2);
    printf("El resultado es %d\n", resultado);
}
```

El ejemplo 4.1 define y utiliza la función `producto` para llevar a cabo la multiplicación de forma externa a `main`. Como vemos, se trata de un paso por valor. Las variables `a` y `b` son locales a la función `producto` y reciben como valor inicial en el momento de la invocación una copia del número almacenado en las variables de `main` `numero1` y `numero2`, respectivamente. Si cambiáramos el valor de `a` o `b` dentro de `producto` no cambiará `numero1` ni `numero2`. El cálculo del producto se deposita en la variable `c` y una copia de su valor es devuelta a `main` mediante la orden `return`.

**¿Jugamos al ahorcado?**

Después del calentamiento del ejemplo 4.1 es hora de pasar a la acción. La mejor manera de entrenar nuestros nuevos conocimientos es mediante un juego. Esta vez hemos elegido uno al que quizá hayáis jugado de niños: el juego del ahorcado. Por cierto, qué título más macabro para un juego infantil.



**Juego 4.1: El juego del ahorcado**

Realiza un programa para jugar al juego del ahorcado con dos jugadores. El primer jugador introduce una palabra y el segundo debe adivinarla. Para ello, deberá introducir letras a través del teclado. Si la letra introducida pertenece a la palabra, se indicará en qué posiciones aparece. En caso contrario, se contabiliza un fallo y se completa ligeramente el dibujo de un muñeco ahorcado. El juego finaliza cuando el segundo jugador adivina la palabra o bien cuando el dibujo está completo.

**Código**

```
#include <stdio.h>

#define LONGMAX 20
#define MAXFALLOS 6
#define TRUE 1
#define FALSE 0

void dibujaahorcado(int fallos)
{
    int i;

    for(i=0;i<100;i++) //borrar pantalla
        printf("\n");

    switch(fallos)
    {
        case 1:
            printf("\n\n\n\n");
            printf (" ___ \n");
            break;

        case 2:
            printf("\n\n\n");
            printf("_|_\n");
```

```
            break;

         case 3:
            printf("\n\n");
            printf("   __\n");
            printf(" | \n");
            printf(" | \n");
            printf(" | \n");
            printf("_|_  \n");
            break;

          case 4:
             printf("\n");
             printf("   __ \n");
             printf(" |   | \n");
             printf(" | \n");
             printf(" | \n");
             printf("_|_ \n");
             break;

          case 5:
             printf("   __ \n");
             printf(" |   |\n");
             printf(" |   o \n");
             printf(" |   | \n");
             printf("_|_   \n");
             break;

          case 6:
             printf("   __ \n");
             printf(" |   | \n");
             printf(" |   o \n");
             printf(" |   | \n");
             printf("_|_ / \\ \n");
             break;

          default:
          break;
   }
}


int main ()
{
```

```
    char palabra[LONGMAX], mascara[LONGMAX], cadena[LONGMAX], letra;
    int i, longitud, acierto, gameover=0, numfallos=0;

    //Jugador 1
    printf("Juego del ahorcado\n");
    printf("Jugador 1 introduce palabra: ");
    gets(palabra); //frases tambien sirven

    //Contar num de letras en palabra
    for(i=0; palabra[i]!='\0'; i++)
        mascara[i]='_';
    longitud=i-1; //indice a la ultima letra

    do
    {
        //Jugador 2
        dibujaahorcado(numfallos);

        printf("La palabra es: "); //imprimir mascara J2
        for(i=0;i<=longitud;i++)
            printf("%c ", mascara[i]);

        printf("\n Jugador 2, elige letra: ");
        gets(cadena); //por si se teclea mas de una letra
        letra=cadena[0];

        for(i=0, acierto=FALSE;i<=longitud;i++)
        {
            if(letra==palabra[i])
            {
                acierto=TRUE;
                mascara[i]=letra;
            }
        }
        if(acierto==FALSE)
        {
            numfallos++;
            if(numfallos==MAXFALLOS)
                gameover=TRUE;
        }
        else
        {
```

```
111             i=0;
112             while(palabra[i]==mascara[i])
113                 i++;
114 
115             if(i>=longitud)
116                 gameover=TRUE; // J2 adivino la palabra
117         }
118     }
119     while(gameover==FALSE);
120 
121     // juego acabo
122     dibujaahorcado(numfallos);
123 
124     if(acierto)
125         printf("\n Enhorabuena!!");
126     else
127         printf("\n Has perdido");
128 
129      printf("\n La palabra era: %s\n", palabra);
130 }
```

Como vemos en el juego 4.1, el código comienza con la preceptiva inclusión de la librería `stdio.h` y una sobredosis de `#define`. Se trata de una buena práctica: cada vez que se usa una etiqueta definida de esta manera se sustituye un número en el código fuente por una etiqueta que tiene un significado fácil de interpretar. Gracias a ello se evita la aparición de números "mágicos" en código, es decir, cifras cuya procedencia o significado pueden ser difíciles de interpretar para todo aquel que no haya realizado el programa. Más aún, al propio autor del código le acabará sucediendo lo mismo con el paso del tiempo.

El programa comienza con la definición de la función auxiliar `dibujaahorcado`, que recibe un número entero donde se indican los fallos que lleva el segundo jugador en su búsqueda de la palabra proporcionada por el primer jugador. Este valor se almacena en la variable local `fallos`. Como se puede ver también en la línea 8, la función no devuelve nada (el tipo devuelto es `void`). El propósito de `dibujaahorcado` es claro: se limpia la pantalla y se dibuja un ahorcado más o menos completo en función del valor de la variable `fallos`. Dado que se contemplan seis posibilidades de dibujo diferente, la estructura de control de flujo condicional `switch` es la opción más apropiada.

Antes de pasar a analizar el código de `main`, conviene reparar en que `dibujaahorcado` se extiende desde la línea 8 a la 64. Integrar semejante cantidad de código dentro de `main` para una tarea tan concreta complicaría el desarrollo del juego en vano. Queda patente, por tanto, la utilidad de las funciones.

La función `main` comienza con la definición de varias cadenas de caracteres, a saber:

- **`palabra`**: se utiliza para recoger la palabra introducida por el primer jugador.

- **`mascara`**: se utiliza para almacenar lo que el jugador dos ve en pantalla. Se utiliza este nombre porque enmascara la palabra que se desea encontrar.

- **`cadena`**: se define para recoger lo que el segundo jugador introduce por el teclado. De esta manera se evitan problemas derivados de que el jugador dos introduzca por error más de una letra, tal y como puede verse en las líneas 93 y 94.

En cuanto al resto de variables, destacan **`longitud`**, que almacena la cantidad de letras de la palabra (o frase) introducida por el primer jugador, **`acierto`**, variable auxiliar que recoge el hecho de que la letra introducida por el segundo jugador esté en la palabra, y **`numfallos`**, que contabiliza el total de fallos hasta el momento.

Entre las líneas 72 y 80 el jugador 1 introduce la palabra y se calcula su longitud a medida que se rellenan los caracteres correspondientes en **`mascara`** con el guión bajo '_'. El bucle del juego (líneas 82 a 119) comienza con el dibujo del ahorcado según el número de fallos actual (invocación de **`dibujaahorcado`** en la línea 85, la impresión de la palabra enmascarada (líneas 87 a 89) y la recepción de la letra introducida por el teclado (líneas 91 a 93). Tras ello, un bucle **`for`** recorre la palabra buscando la letra introducida en ella y desvelando posiciones tras la máscara de guiones bajos si es preciso. La variable **`acierto`** valdrá 1 al salir del bucle si la letra se encontró en la palabra y 0 en caso contrario (líneas 95 a 102). En función de este valor se pueden incrementar los fallos del jugador (líneas 103 a 108), que podrían llevar incluso al final del juego, o se comprueba si tras el último acierto se completó la palabra (líneas 110 a 117). El juego finaliza imprimiendo el ahorcado en su estado actual por última vez (línea 122) y un mensaje de enhorabuena o fracaso en función de si la última letra acabó o no en fallo (líneas 124 a 129). (Evidentemente el juego acaba bien para el segundo jugador cuando en la última jugada acierta.)

En la figura 4.1 se muestra una captura de pantalla de nuestro juego del ahorcado en ejecución.

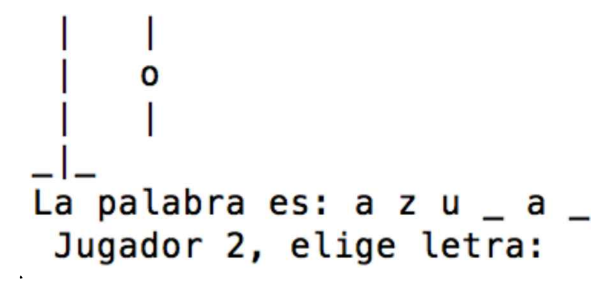



**Figura 4.1:** Pantalla del juego 4.1.

### 4.2.2. Paso por referencia: trabajando con originales

Cuando se trabaja con vectores o matrices como argumentos de entrada, la función recibe el original y no una copia de la información como en el caso anterior. En este caso, la variable local, es decir, el nombre con el que se ha definido la matriz o vector en la función, se convierte en un *alias* de la matriz o del vector original. Por tanto, cualquier cambio que experimente el vector o la matriz dentro de la función afecta al resto del programa, debido a que función invocadora e invocada comparten la zona de memoria de estas variables. Esto proporciona también una alternativa a `return` para la devolución de información al exterior.

#### Indagando en el paso por referencia: punteros

Como dijimos en el primer capítulo, C es un lenguaje de nivel medio y ello implica la existencia de características de bajo nivel como los punteros, concepto que se introduce a continuación y que suele ser espinoso para los programadores novatos.

Un puntero se define como una variable cuyo contenido es la dirección de memoria de otra variable. Los punteros son tan característicos de C que hasta tienen sus propios tipos de dato, como veremos en breve. De momento, pensemos en el siguiente ejemplo: supongamos que tenemos una variable almacenada en la dirección de memoria 3500 llamada `numero` y otra variable más a la que, en un alarde de originalidad, llamaremos `puntero`. Para que `puntero` se convierta en un puntero de `numero` deben cumplirse dos condiciones:

1. Que el contenido de la variable `puntero` sea la dirección de la variable `numero`, esto es, 3500.
2. Que el tipo de dato de `puntero` sea puntero al tipo de dato de la variable `numero`. Por tanto, si `numero` es una variable de tipo entero, `puntero` debe ser de tipo puntero a entero. Si `numero` fuera un flotante, `puntero` debería ser de tipo puntero a flotante.



**Recuerda 4.3: Operadores útiles para trabajar con punteros**

La utilización de punteros está muy ligada a los siguientes operadores:

- Operador proporcionar dirección: se representa con el carácter ampersand '&'. Cuando precede a una variable se asocia a ella para proporcionar su dirección.
- Operador proporcionar contenido: se representa con el carácter asterisco '*'. Cuando precede a una dirección de memoria se asocia a ella para proporcionar la información contenida en dicha dirección.

Siguiendo con el ejemplo, supongamos que **numero** vale 2 y recordemos que se encuentra almacenada en la dirección 3500. Escribir en C **&numero** es equivalente a escribir 3500, ya que el operador **&** proporciona la dirección de la variable a la que acompaña. Por tanto, para hacer que la variable **puntero** *apunte* a **numero** basta con realizar la asignación **puntero=&numero**. Hecho esto, pueden utilizarse el operador ***** y la variable **puntero** para acceder a la información existente en el interior de la dirección 3500. Para ello, basta con escribir ***puntero**. Lo más notable es que esto sirve tanto para leer como para escribir en el interior de esa dirección de memoria. Por ejemplo, si ejecutamos la asignación ***puntero=5** el contenido de la dirección 3500 pasa a ser 5, es decir, ¡esta instrucción cambia el valor de la variable **numero** de forma externa a ella!

La segunda de las dos condiciones enunciadas anteriormente impone una necesidad de compatibilidad entre los tipos de dato de las variables **numero** y **puntero**. El motivo para ello es que la variable **numero** puede ocupar más de una posición de memoria en función de su tipo de dato. Por ejemplo, un dato de tipo **float** ocupa más que uno de tipo **int**, y a su vez este ocupa más que uno de tipo **char**. Para que a través de ***puntero** pueda accederse al contenido de la variable **numero** es preciso que la variable **puntero** tenga asociado el número de posiciones de memoria que ocupa **numero**, o de lo contrario no podrá recuperar su contenido. Es decir, para que el operador ***** funcione bien, es imprescindible que el tipo de dato de la variable apuntada esté en consonancia con el del puntero que la apunta.

Dicho esto, ya estamos en condiciones de enseñaros cómo se declaran los punteros en C.



**Recuerda 4.4: Declaración de punteros en C**

Un puntero que apunte a un dato de tipo `tipo_de_dato` se declara como se indica a continuación.

**Código**

```
tipo_de_dato *nombre_variable_puntero;
```

El tipo de dato del puntero es **tipo_de_dato ***, lo que se lee como puntero a **tipo_de_dato**, donde **tipo_de_dato** indica el tipo de dato al que el puntero apuntará (**int**, **float**...). De alguna manera la propia sintaxis de la declaración establece este hecho, ya que ***nombre_variable_puntero** representa el contenido de una posición de memoria en la que se almacena una variable con tipo igual a **tipo_de_dato**.

Por ejemplo, supongamos las declaraciones siguientes:

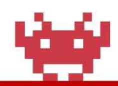
```
int *puntero_a_entero; char *ptr_a_char;
```

La variable **puntero_a_entero** es de tipo puntero a entero y la variable **ptr_a_char** es de tipo puntero a char. Asimismo, esto se puede interpretar también como que

***puntero_a_entero** es equivalente a tener una variable de tipo entero y ***ptr_a_char** es equivalente a una variable de tipo carácter.

Ahora que ya conocemos los punteros podemos desvelar un hecho que hemos mantenido en secreto hasta este momento: habéis trabajado con punteros todo este tiempo sin saberlo. El nombre de un vector o una matriz es un puntero al comienzo de la zona que ocupa en memoria. Este hecho tiene una consecuencia importante: cuando se pasa un vector o una matriz como parámetro no se transmite otra información que la de la dirección de comienzo de la zona de memoria en la que se almacena. El vector o la matriz utilizado como variable de entrada en la función recibe este valor y pasa a apuntar a la misma zona de memoria, lo que hace que cualquier cambio en un elemento del vector o la matriz permanezca en memoria aun cuando la función termine, ya que se está modificando una zona de memoria ajena a la función.

Por todo lo anterior, el paso por referencia puede concebirse también como un caso particular de paso por valor en el que lo que se transmite es una copia de una dirección de memoria. De hecho, esto se aprovecha en el siguiente ejemplo para hacer que una función modifique el valor de dos variables externas a ella.



**Ejemplo 4.2: Paso por referencia**

Realiza un programa que utilice una función para intercambiar el contenido de dos variables enviadas por referencia.

**Código**

```
#include <stdio.h>

void intercambia(int *pa, int *pb)
{
    int aux;
    aux=*pa;
    *pa=*pb;
    *pb=aux;
}

int main()
{
    int a=2;
    int b=5;
    printf("\n a vale %d y b vale %d", a, b);
    intercambia(&a, &b);
    printf("\n a vale %d y b vale %d \"n, a, b);
}
```

El enunciado del ejemplo 4.2 especifica que el paso de información a la función es por referencia, esto es, mediante direcciones de memoria. La función `intercambia` utiliza el operador `*` para acceder al contenido de las dos variables referenciadas e intercambiar su valor mediante la ayuda de la variable auxiliar `aux`.

Este ejemplo no podría llevarse a cabo sin el paso por referencia. Una función que recibiera por valor las variables `a` y `b` podría intercambiar el contenido de las copias pero nunca de los originales. Cuando se quiere modificar el valor de una variable de otra función de nada sirve saber su contenido. Conocer su dirección, en cambio, da la oportunidad de ir a buscarla en memoria tanto para leer su valor como para asignarle uno nuevo. Así que cuidado, variables, sabemos donde vivís.

Finalmente, conviene señalar que los punteros nos dan mucho poder dentro de la memoria pero también generan una gran responsabilidad. Un fallo en la programación podría hacer que un puntero apuntara a una zona de la memoria externa al propio programa. Una operación de lectura o escritura en dicha zona puede tener consecuencias desagradables, principalmente porque el sistema operativo o el antivirus podrían identificar el programa como un agresor potencial dentro de la máquina. Como poco, esto llevará a fallos en la ejecución e incluso a que el programa se cierre de forma inesperada, así que mucho cuidado a la hora de utilizar los punteros.

## 4.3. La función main

La función `main` se define de la misma forma que cualquier función en C, pero tiene una peculiaridad lo suficientemente importante como para dedicarle una sección entera: hay que definirla siempre. La función `main` debe encontrarse en todos los programas escritos en lenguaje C, sin excepciones. Define, valga la redundancia, la función principal del programa, esto es, su cometido. Todo programa está hecho para resolver un problema y `main` se encarga de *dirigir* el algoritmo que da solución al mismo. Dediquemos unos instantes a reflexionar sobre ello.

Existen multitud de problemas susceptibles de ser resueltos con un programa de ordenador. El número de líneas de código necesarias para ello depende, como es lógico, de la complejidad del problema. En general, cuanto más complicado sea el problema, mayor será la extensión del código. Un programa extremadamente complejo, por ejemplo, un sistema operativo, puede estar compuesto de millones de líneas de código.

He aquí la pregunta del millón (de líneas de código), ¿se escriben todas estas líneas dentro de la función `main`? Es evidente que un programa así escrito sería imposible de depurar o ampliar. Justo en este punto es donde entran las funciones de las que hablábamos en los apartados anteriores. Un problema complejo puede ser descompuesto en muchos problemas pequeños, cada uno de los cuales puede ser resuelto por una función. El papel de `main` es el de coordinar (invocar) a las distintas funciones para que el programa lleve a cabo su cometido con éxito. En el ejemplo del restaurante que veíamos, `main` es el jefe del negocio. En una orquesta de música, `main` sería el director, el responsable de que todos toquen a su debido tiempo.

La división de un programa en funciones está muy vinculada con el concepto de modularidad, que consiste en dividir un programa en pequeños módulos que interactuan entre sí, lo que simplifica su diseño, depuración y mantenimiento. Esta es justo la razón de ser de las funciones: descomponer en pequeñas partes la solución del problema propuesto. Dado que el código de cada función es independiente del resto del programa, su funcionamiento correcto podrá ser validado de forma aislada. Esto reduce enormemente el esfuerzo de depuración, ya que se tarda mucho menos en revisar diez funciones de veinte líneas que un programa de doscientas.

Una vez convencidos de las ventajas de dividir un programa en funciones, debemos advertiros del peligro de crear un número excesivo de ellas. Casi tan pernicioso es un programa que tiene demasiadas funciones con muy pocas líneas de código como uno que tiene solamente un `main` sobrecargado. La virtud, como siempre, está en el término medio. La creación de funciones debe guiarse por el principio de máxima cohesión y mínimo acoplamiento. Cada una debe tener un propósito claro y ser lo más independiente posible del resto del programa. El sentido común y la experiencia acumulada como programadores os ayudarán a descomponer el código en funciones de forma equilibrada. No obstante, y como criterio orientativo para los programas que encontraréis en el libro, siempre que `main` tenga más de treinta o cuarenta líneas de código será conveniente introducir alguna función extra que realice parte del trabajo, especialmente si es posible identificar una parte del código muy relacionada con un único propósito (y, por tanto, candidata a ser encapsulada en una función).

## 4.4. Recetas para programas con funciones

Acabamos de ver los ingredientes funcionales que componen un programa en C. En esta sección aprenderemos cómo se deben combinar para crear un programa y presentaremos dos recetas que definen la estructura que debe seguir el código para que pueda ser compilado. La principal diferencia entre ambas es la posición de `main` dentro del código. Prestad atención al orden en el que aparecen los elementos, puesto que en este caso el orden de los factores es importante.

### 4.4.1. Receta estándar



**Recuerda 4.5: Receta estándar para programar con funciones**

1. Directivas del preprocesador.
2. Definición de funciones auxiliares.
3. Definición de la función `main`.

Este esquema se corresponde con el que ya vimos en el ejemplos 4.1 y 4.2, por lo que no se introducirá ningún código adicional. Con todo, sí que conviene dedicar unas líneas a la directiva **#include** ahora que ya conocemos las funciones.



**Recuerda 4.6: Uso de librerías en C**

La inclusión de una librería de funciones externa en un programa de C se realiza con la directiva del preprocesador **#include**.

**Código**

```
# include <nombre_biblioteca>
```

La directiva **#include** incluye en nuestros programas código en C extraído de unos archivos externos conocidos como bibliotecas o librerías[1] y debería despertar en todos nosotros un fuerte sentimiento de gratitud por todo el trabajo que nos ahorra. Imaginad por ejemplo que tuviéramos que escribir en cada programa *todo* el código necesario para leer caracteres del teclado o imprimirlos en pantalla a bajo nivel. En este momento no sabríais cómo hacerlo, pero no suena sencillo, ¿verdad? Gracias a esta directiva, basta escribir `#include <stdio.h>` para que todo este esfuerzo se reduzca a usar funciones como `printf` o `scanf`. Es decir, `include` *incluye* funciones auxiliares externas dentro de nuestro código justo antes de la compilación, ahorrándonos con ello una enorme cantidad de trabajo. Existen gran cantidad de librerías externas que pueden ser incluidas en un programa y con el tiempo llegaréis a conocer muchas. La más famosa de todas en C es, cómo no, stdio.h, cuyo nombre viene del inglés "standard input/output" (entrada/salida estándar). Es decir, se trata de una biblioteca que aporta al programa funciones relacionadas con la entrada y salida estándar del ordenador (el teclado y la pantalla).

### 4.4.2. Receta con declaraciones



**Recuerda 4.7: Receta con declaraciones**

1. Directivas del preprocesador.
2. Declaración de funciones auxiliares.
3. Definición de la función `main`.
4. Definición de funciones auxiliares.

[1] El término en inglés es *library*, cuya traducción es biblioteca. Sin embargo, ha sido traducido como *librería* el suficiente número de veces como para que el error sea ya irreparable.

Como puede verse, esta receta cambia ligeramente la disposición de los elementos dentro del código e introduce el concepto de declaración de función. Está diseñada para el caso en que prefiramos definir `main` antes que el resto de las funciones del programa. En este caso, conviene imaginar el programa como una orquesta de músicos. No cabe duda de que si un programa fuera una orquesta, `main` sería el director y el resto de las funciones serían los músicos. Lo mínimo que pide el director de la orquesta antes de empezar a actuar es conocer los músicos que tendrá a su disposición. Esto es justamente lo que hace la declaración, proporcionar datos importantes sobre la forma en la que la función interactúa con el resto del programa. Pero, ¿no era eso justo lo que se indicaba en la primera línea de la definición de una función? (Véase el cuadro para recordar 4.1 en el inicio del capítulo.) Por ello, la sintaxis de la declaración es exactamente la misma que la de la primera línea de la definición de la función, aunque opcionalmente se permite omitir el nombre que reciben los argumentos de entrada de la función, pues en la declaración solo interesan sus tipos de dato.



**Recuerda 4.8: Declaración de funciones en C**

La declaración de una función en C se realiza mediante la inclusión de su prototipo antes de la definición de `main`. La sintaxis del prototipo es:

**Código**

```
Tipo_de_dato_salida  Nombre_función (tipo_de_dato_entrada_1,
tipo_de_dato_entrada2, …);
```

El cuadro para recordar 4.8 muestra que la declaración de una función consiste en su caracterización indicando nombre y tipo de argumentos de entrada y salida. Ojo con lo último, ¡las funciones son muy quisquillosas con la información que se introduce en su prototipo! Asimismo, dicha información debe coincidir exactamente con la que se proporcione en la definición de la función. Veamos un ejemplo.



**Ejemplo 4.3: Multiplicador con declaración de función**

Realiza un programa que pida al usuario dos números por teclado y los multiplique utilizando una función. El programa debe realizarse conforme a la receta del cuadro para recordar 4.7.

**Código**

```
#include <stdio.h>

int producto(int, int);
```

```
int main ()
{
    int numero1;
    int numero2;
    int resultado;

    printf("Introduzca el primer numero: ");
    scanf("%d",&numero1);
    printf("Introduzca el segundo numero: ");
    scanf("%d",&numero2);

    resultado=producto(numero1, numero2);

    printf("El resultado es %d\n", resultado);
}

int producto(int a, int b)
{
    int c;
    c=a*b;
    return (c);
}
```

El ejemplo 4.3 es una variación del 4.1. Ambos códigos llevan al mismo programa, pero el orden en el que están escritos varía. Nótese que la declaración de la función de la línea podría haberse realizado también escribiendo el nombre de los argumentos, esto es, **`int producto(int a, int b);`**.

Como puede verse también, la nueva receta denota una peculiaridad relacionada con la función **`main`**: **`main`** no se declara, simplemente se define. Se sabe que va a existir siempre, así que su declaración sería superflua. Examinando ambas recetas se ve que **`main`** determina el lugar en que las funciones se declaran o se definen en un programa en C. El resto de funciones gira en torno a ella, literalmente.

Tras estos ejemplos, estamos en disposición de pasar a lo que más nos gusta: ¡programar juegos!

## 4.5. Juegos con funciones

En esta sección recopilamos algunos juegos en los que se utilizan funciones con objeto de asentar los conocimientos del capítulo. En primer lugar desarrollaremos una versión simplificada del popular *Hundir la Flota*. Tras ello, crearemos una peculiar versión en C del famoso *Angry Birds*.

### 4.5.1. Hundir la flota

El nombre Milton Bradley os resultará desconocido, pero seguro que su legado no. En 1860 fundó una compañía, la Milton Bradley Company, más conocida por sus siglas MB, que fabrica y comercializa juegos de mesa tan populares como *¿Quién es Quién?*, *Operación* o *Cocodrilo Sacamuelas.* El juego que protagoniza esta subsección fue lanzado por MB en 1931 con el nombre original de *Battleship* y fue comercializado en España como *Hundir la Flota*, pese a que la traducción original del título estaría más cerca de *Batalla Naval.* Paradójicamente, la película inspirada en el juego que se lanzó en 2012, y que contó en su reparto con actores tan conocidos como Liam Neeson, sí retuvo en su adaptación española el nombre original de *Battleship*, posiblemente por el miedo a las consecuencias que pudiera tener en taquilla que se asociara el film con el juego de mesa.

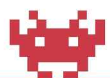
Por si algún lector desconociera el juego, explicaremos aquí su mecánica. *Hundir la Flota* es un juego para dos jugadores en el que cada jugador dispone de un tablero rectangular dividido en casillas que ocultará a su oponente durante el transcurso de la partida. Cada jugador coloca en su tablero una serie de barcos de guerra y dispara en su turno a una casilla del tablero rival. Si el disparo falla, el jugador contrario dirá *agua*; si el disparo alcanza una casilla ocupada por uno de los barcos, dirá *tocado*; finalmente, si los disparos del jugador han tocado todas las casillas que ocupa un barco, dirá *tocado y hundido.* Gana el jugador que acabe antes con todos los barcos del rival. Generalmente las filas del tablero se identifican con letras del abecedario y las columnas con números para simplificar la localización de cada casilla. Asimismo, también es habitual que cada jugador disponga de un segundo tablero para anotar las posiciones en las que ya ha disparado.

En este apartado desarrollaremos una versión en C de *Hundir la Flota.* Será nuestro primer juego complejo porque estará compuesto de varias funciones. De este modo observaremos todo el potencial que las funciones ofrecen en el desarrollo de programas. No obstante, la versión del juego que implementaremos será una versión simplificada. En primer lugar, cada jugador dispondrá únicamente de un tablero, ya que será el computador el que se encargará de mostrar por pantalla la información relacionada con los disparos ya efectuados. Asimismo, jugaremos únicamente con barcos que ocuparán una casilla del tablero.

A continuación, desarrollaremos a modo de ejemplos de programación todas las funciones que utilizaremos en el programa.

#### Función para limpiar la pantalla

En el transcurso del juego será necesario actualizar el estado del tablero, lo que se hará borrando y reimprimiendo. Veamos una función para “borrar” la pantalla, aunque la función no borra nada en realidad. Simplemente imprime nuevas líneas de texto con `printf` un número suficiente de veces como para generar el efecto de borrado.



**Ejemplo 4.4: Hundir la flota: borrar pantalla**

Comencemos con una función muy sencilla que limpia la pantalla imprimiendo el carácter de nueva línea cien veces.

**Código**

```
void limpiapantalla()
{
    int i;
    for(i=1; i<=100; i++)
        printf("\n");
}
```

La función del ejemplo 4.4 no recibe ni devuelve información, tal y como puede verse en su primera línea. Por ello, el conjunto de argumentos de entrada aparece vacío y el tipo de dato de salida es `void`.

### Función para convertir letras en números

Para ser fieles al juego original, identificaremos las filas con letras y las columnas con números. Dado que C identifica las posiciones dentro de una matriz con números, es necesario transformar la letra de la fila.



**Ejemplo 4.5: Hundir la flota: de letra a número**

Desarrolla una función que reciba un carácter que identifica a una fila del tablero y devuelva el número correspondiente a dicha fila en una matriz de C. Asimismo, la función devuelve -1 si la letra introducida corresponde a una fila fuera del tablero o a un carácter incorrecto.

**Código**

```
int transformaletra(char letra)
{
    int resultado=-1;

    /*Si la letra es mayúscula...*/
    if(letra>='A'&&letra<='A'+NUMFILAS-1)
        resultado=letra-'A';

```

```
    /*Si la letra es minúscula... */
    if(letra>='a'&&letra<='a'+NUMFILAS-1)
        resultado=letra-'a';

    return(resultado);
}
```

La clave para entender la transformación realizada en el ejemplo 4.5 es que los caracteres se codifican en memoria como números según el código ASCII. Por ejemplo, la letra ‘A’ se codifica con el número 65. De este modo, es posible *restar* letras, ya que lo que realmente se están restando son los números con los que están codificadas en memoria.

### Función para inicializar el tablero

La siguiente función inicializa los tableros colocando un cero en cada una de sus casillas. Se trata de una función que no devuelve nada y que recibe como argumento una matriz.



**Ejemplo 4.6: Hundir la flota: inicializar el tablero**

Desarrolla una función que reciba una matriz como argumento de tamaño NF×NC, asumiendo que ambos tamaños están definidos mediante la directiva del preprocesador #define.

**Código**

```
void inicializatablero(int m[NF][NC])
{
    int i,j;
    for(i=0; i<NF; i++)
        for(j=0; j<NC; j++)
            m[i][j]=0;
}
```

Debe tenerse en cuenta que las matrices siempre aparecen con sus tamaños dentro de la definición de la función. En este ejemplo se supone que los tamaños están definidos como dos constantes numéricas, `NF` y `NC`, introducidas mediante `#define`.[2]

[2] Estrictamente, solamente haría falta introducir el número de columnas en la definición de la función. No obstante, en este libro especificaremos tanto el número de filas como el de columnas con objeto de mantener la uniformidad con la manera en la que se declaran las matrices en el programa.

Otro aspecto a tener en cuenta acerca del ejemplo 4.6 es que en C se trabaja siempre por referencia cuando se trata de vectores o matrices. Es decir, aunque pueda parecer que la matriz `m` es local a la función `inicializatablero`, realmente se está inicializando la matriz que se haya pasado como argumento al invocar a la función.

### Función para detectar el fin del juego

En cada turno debe comprobarse si se ha llegado al final de la partida. El juego termina si no hay barcos a flote en el tablero. Para comprobarlo programaremos la función `gameover`.



**Ejemplo 4.7: Hundir la flota: detección de fin del juego**

Desarrolla una función que recibe una matriz y devuelve un 1 en caso de que el juego haya concluido y un 0 en caso contrario. Dado que codificamos los barcos como números positivos, basta con recorrer la matriz hasta encontrar uno para saber si el juego debe continuar. Si no se encuentra, es que el juego ha terminado.

**Código**

```c
int gameover(int m[NF][NC])
{
    //Hipótesis: NO hay barcos -> resultado=1
    int i,j,resultado=1;

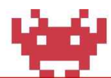
    //Se itera sólo mientras la hipótesis se mantiene
    for(i=0; i<NF&&resultado; i++)
        for(j=0; j<NC&&resultado; j++)
            if(m[i][j]>0)
                resultado=0;

    return(resultado);
}
```

Nótese que se ha realizado la función del ejemplo 4.7 asumiendo que no hay barcos en el tablero. De este modo, puede dejar de recorrerse la matriz en cuanto se encuentre un barco. Si se hubiera realizado la hipótesis opuesta, es decir, que hay barcos, habría que recorrer siempre toda la matriz hasta estar seguros de que no hay ningún barco en el tablero. Desde un punto de vista computacional, la solución propuesta es más eficiente por este motivo.

**Función para introducir los barcos**

Otra función que nos resultará de utilidad será una que permita a cada jugador introducir los barcos en su tablero. La función `coloca` recibe como argumento una matriz con el tablero de un jugador y le pregunta por las posiciones de sus barcos. Se asume que el número de barcos está definido mediante la directiva `#define` en la constante `NB`.



**Ejemplo 4.8: Hundir la flota: colocación inicial de barcos**

Desarrolla una función que recibe una matriz con el tablero y pregunta a cada jugador por la posición inicial de cada uno de sus NB barcos.

**Código**

```
1  void coloca(int m[NF][NC])
2  {
3      char letra;
4      int i, j, l;
5
6      for(l=1; l<=NB; l++)
7      {
8          printf("Iteracion %d\n",l);
9          printf("Intro fila (A-%c):", 'A'+NF-1);
10         scanf(" %c", &letra);
11         i=transformaletra(letra);
12         printf("Intro col (1-%d):",NC);
13         scanf(" %d", &j);
14         j=j-1;
15         m[i][j]=1;
16     }
17 }
```

En este ejemplo de programación merece la pena prestar atención a la línea 9, que muestra información al usuario acerca del rango de letras disponible para la fila. En dicha línea `printf` evalúa una expresión para mostrar por pantalla la letra de la última fila del tablero. Asimismo, desde esta función se invoca a `transforma` en la línea 11, función que programamos anteriormente. Como vemos, las funciones auxiliares que incluyamos en nuestros programas podrán ser invocadas desde otras funciones diferentes a `main`.

Finalmente, también se debe reparar en que los `scanf` de las líneas 10 y 13 llevan un espacio al inicio, justo antes del carácter '%'. Con este truco se evitan problemas derivados de la pulsación de la tecla *enter* tras la introducción por teclado de la letra

o del número por parte del usuario. Para entender bien el truco es necesario introducir el concepto de *buffer* del teclado. Todo lo que el usuario teclea es almacenado provisionalmente en una especie de archivo interno llamado *buffer*, donde permanece hasta que algún programa acude a recoger la información.[3] Cuando el usuario introduce la letra o el número y pulsa *enter*, **`scanf`** se lleva la información útil y deja dentro del *buffer* la pulsación de *enter*. Cualquier información adicional que el usuario escriba por teclado quedará depositada tras el *enter* "olvidado". Así, la siguiente llamada a **`scanf`** se encuentra para empezar con que el usuario ha pulsado *enter*, por lo que puede interpretar erróneamente que el usuario no quiso introducir información adicional. Mediante la adición del espacio, se le indica a **`scanf`** que antes de la información útil puede encontrar algún caracter "invisible" (como el *enter* olvidado), lo que evita el problema.

### Función para imprimir el tablero

El programa mostrará el tablero de ambos jugadores, indicando qué casillas han recibido un disparo. La función `imprimetablero` desarrolla esta labor.



**Ejemplo 4.9: Hundir la flota: impresión del tablero**

Desarrolla una función que recibe una matriz con el tablero y lo muestre por pantalla. En concreto, deberá representarse un asterisco en cada casilla donde ya se haya disparado y una interrogación donde aún no se haya disparado.

**Código**

```
void imprimetablero(int m[NF][NC])
{
    int i,j;
    for(i=0; i<NF; i++)
    {
        for(j=0; j<NC; j++)
            if(m[i][j]<0)
                printf("*");
            else
                printf("?");

        printf("\n");
    }
}
```

[3] Si alguna vez habéis tecleado dentro de una aplicación y no ha aparecido nada en pantalla hasta unos segundos después, el programa que estabais utilizando ha tardado en acceder al *buffer*, posiblemente porque el procesador estaría ocupado en otras tareas durante esos momentos.

### Función turno

Hemos desarrollado funciones para casi todo menos para lo más importante, ¡los turnos de los jugadores! La función `turno` recibe la matriz con el tablero rival y solicita al usuario que introduzca una posición del mismo para realizar un disparo.



**Ejemplo 4.10: Hundir la flota: turno de jugador**

Desarrolla una función que recibe una matriz con el tablero rival y pida al jugador con el turno actual que elija dónde disparar.

**Código**

```
1  void turno(int m[NF][NC])
2  {
3      char letra;
4      int i, j;
5      printf("Intro fila (A-%c):", 'A'+NF-1);
6      scanf(" %c", &letra);
7      i=transformaletra(letra);
8      printf("Intro col (1-%d):",NC);
9      scanf(" %d", &j);
10     j=j-1;
11     limpiapantalla();
12     if(m[i][j]>0)
13         printf("Barco hundido!\n");
14     else
15         printf("Agua\n");
16     m[i][j]=-1;
17     imprimetablero(m);
18     printf("\nPulsa enter para acabar el turno.");
19     letra=getc(stdin); //para enter previo
20     letra=getc(stdin); // para enter nuevo
21 }
```

La función del ejemplo 4.10 solicita la información de manera convencional y muestra al usuario información acerca de su disparo, imprimiendo para ello los mensajes “Agua” o “Barco hundido!” (líneas 12 a 16). Tras ello, imprime de nuevo el tablero con el disparo ya efectuado (línea 17) y pide al usuario que pulse *enter* para terminar. Debido al problema anteriormente indicado en el ejemplo 4.8, es necesario realizar dos veces la invocación a la función `getc`. La primera vez la función recoge del *buffer* del teclado la pulsación de *enter* olvidada por el `scanf` de la línea 9. La siguiente invocación (línea 20) recoge la pulsación de *enter* que da lugar al siguiente turno.

## Y, por fin, la función principal

Tras definir todas las funciones necesarias para realizar el programa, podemos realizar el código de la función principal o `main`, que será la encargada de dirigir al resto de funciones. Observad en el siguiente código la relativa facilidad con la que se entiende lo que hace el programa gracias al uso de las funciones.



**Juego 4.2: Hundir la flota**

Desarrolla el juego Hundir la Flota a partir de las funciones anteriormente desarrolladas.

**Código**

```
#include <stdio.h>
#define NF 8
#define NC 8
#define NB 3

// Funciones auxiliares
void limpiapantalla();
int transformaletra(char letra);
void inicializatablero(int m[NF][NC]);
int gameover(int m[NF][NC]);
void coloca(int m[NF][NC]);
void imprimetablero(int m[NF][NC]);
void turno(int m[NF][NC]);

// Función principal
int main()
{

    int findeljuego=0;   //si valiera 1, se acaba el juego
    int J1[NF][NC];      //Tablero del primer jugador
    int J2[NF][NC];      //Tablero del segundo jugador

    //Inicialización de los tableros
    inicializatablero(J1);
    inicializatablero(J2);

    //Colocación de los barcos en el tablero
    printf("J1 Coloca barcos: \n");
```

```
29      coloca(J1);
30      limpiapantalla();
31
32      printf("J2 Coloca barcos: \n");
33      coloca(J2);
34      limpiapantalla();
35
36      //Bucle principal del juego
37      while(findeljuego==0)
38      {
39          //Turno de J1, que juega solo si no ha perdido
40          if(gameover(J1)==0)
41          {
42              limpiapantalla();
43              printf("Turno J1:\n");
44              imprimetablero(J2);
45              turno(J2);
46              if(gameover(J2))
47              {
48                  findeljuego=1;
49                  printf("\nHa ganado J1!\n");
50              }
51          }
52
53          // Turno de J2, que juega solo si no ha perdido
54          if(gameover(J2)==0)
55          {
56              limpiapantalla();
57              printf("Turno J2:\n");
58              imprimetablero(J1);
59              turno(J1);
60              if(gameover(J1))
61              {
62                  findeljuego=1;
63                  printf("\nHa ganado J2!\n");
64              }
65          }
66      }
67  }
```

En el código del juego 4.2 se han omitido las funciones anteriormente desarrolladas, que deberían depositarse a continuación de `main` en el mismo orden en el que se han declarado (líneas 7 a 13). El programa comienza declarando una variable auxiliar para controlar el fin del juego (línea 19) y los dos tableros (líneas 20 y 21), que

son inicializados (líneas 23 y 24) y usados como argumentos para que los jugadores coloquen sus barcos (líneas 28 a 34). A partir de ahí, un bucle `while` mantiene el juego en ejecución mientras `findeljuego` valga cero. Las operaciones a realizar en cada iteración son sencillas: si el jugador 1 no ha perdido (línea 40), se le imprime su tablero y se le pide que dispare (líneas 42 a 45). Salvo que con su disparo termine con el juego (líneas 46 a 50), se le pasa el turno al siguiente jugador, para el que se repetirán estos mismos pasos (líneas 54 a 65). Finalmente, en la figura 4.2 se muestra el aspecto de nuestro Hundir la Flota en ejecución.

```
Turno J1:
??*?????
????????
??????*?
????????
????????
????????
?*??????
????????
Intro fila (A-H):E
Intro col (1-8):5_
```

**Figura 4.2:** Pantalla del juego 4.2.

## 4.5.2. Angry asteriscos

A falta de pájaros (y licencia) para implementar el fantástico juego de teléfonos móviles *Angry Birds*, creado por *Rovio Entertainment* en 2009, desarrollaremos un homenaje al mismo en el que lanzaremos asteriscos en busca de venganza. Para ello, necesitamos calcular la trayectoria que seguirá nuestro asterisco, que será lanzado con un ángulo y una velocidad introducidos por el teclado con el fin de acertar a una diana ubicada en una posición fija de la pantalla.

La programación de este juego exige calcular la solución de las ecuaciones que rigen el movimiento de nuestro proyectil. Por tanto, necesitamos unos conocimientos básicos de física. De hecho, la simulación de procesos físicos es muy habitual en videojuegos. La lista de ejemplos es interminable: los hilarantes ataques con bazuka en el *Worms*, el movimiento del balón en *FIFA*, los coches de *Gran Turismo*... Todos tienen en común la simulación de leyes físicas en el ordenador para que lo que se muestra por pantalla parezca real.

Por simplicidad, estudiaremos nuestro problema en dos dimensiones. El eje de abscisas será utilizado para representar el movimiento horizontal y el de ordenadas para el vertical. Supondremos que el cañón situado en el origen, es decir, en el punto

(0,0), y llamaremos $x$ e $y$ a las coordenadas del punto en el que se encuentra el asterisco.

Las ecuaciones diferenciales que rigen la evolución de la posición del asterisco se presentan a continuación. En primer lugar, se tiene la aceleración en ambos ejes:

$$a_y(t) = \frac{d^2y}{dt^2} = -g, \quad a_x(t)\frac{d^2x}{dt^2} = 0.$$

Como vemos, únicamente consideramos el efecto de la gravedad ($g$) actuando sobre el eje $y$. Esto implica que la velocidad que experimentará el proyectil en cada dirección será:

$$v_y(t) = \frac{dy}{dt} = -g \cdot t + v_y(0), \quad v_x(t) = \frac{dx}{dt} = v_x(0),$$

donde $v_x(0)$ y $v_y(0)$ son, respectivamente, las velocidades iniciales del proyectil en cada uno de los ejes del problema. Para un proyectil impulsado con velocidad inicial $v_0$ con un ángulo $\alpha$, se tiene $v_x(0) = v_0 \cdot cos(\alpha)$ y $v_y(0) = v_0 \cdot sen(\alpha)$.

Finalmente, la posición del proyectil puede calcularse como:

$$y(t) = -\frac{g \cdot t^2}{2} + v_y(0)t + y(0), \quad x(t) = v_x(0) \cdot t + x(0),$$

donde $x(0)$ e $y(0)$ representan la posición inicial del asterisco, ubicada en el origen como ya dijimos anteriormente. Por tanto, $x(0) = 0$ y $y(0) = 0$.

Una vez desarrolladas las ecuaciones de que determinan la posición del asterisco en función del tiempo, estamos en disposición de implementar el programa. Sin duda, no se trata de las ecuaciones más realistas que podrían utilizarse. Por ejemplo, en nuestro modelo no aparece el efecto del rozamiento del proyectil con el aire. No obstante, como veremos, el resultado en pantalla es razonablemente realista pese a la sencillez del modelo empleado.



**Juego 4.3: Angry asteriscos**

Desarrolla un juego en el que el jugador introduzca por teclado la velocidad inicial y el ángulo de disparo de un asterisco que debe impactar en una diana ubicada en la pantalla. El juego no termina hasta que el jugador acierta.

**Código**

```
#include<stdio.h>
#include<math.h>

#define NUMFILAS 16
#define NUMCOLS 70

```

```
// funcion que inicializa lo que se muestra por pantalla
void iniciapantalla(char pantalla[NUMFILAS][NUMCOLS])
{
    int i, j;
    for(i=0; i<NUMFILAS; i++)
        for(j=0; j<NUMCOLS; j++)
            pantalla[i][j]=' ';

    //dibujo diana
    i=(int)NUMFILAS/2-1;
    pantalla[i-1][NUMCOLS-1]='X';
    pantalla[i][NUMCOLS-1]='X';
    pantalla[i+1][NUMCOLS-1]='X';

    //dibujo cañon
    pantalla[NUMFILAS-1][0]='/';
}

// funcion que imprime la matriz pantalla
void dibujapantalla(char pantalla[NUMFILAS][NUMCOLS])
{
    int i, j;

    printf("\n");

    for(i=0; i<NUMFILAS; i++)
    {
        for(j=0; j<NUMCOLS; j++)
            printf("%c", pantalla[i][j]);

        printf("\n");
    }
}


// funcion que calcula la posicion del objeto
void posicion(float v, float alfa, float t, float *px, float *py)
{
    float pi=3.141592;
    float g=9.81;
    float vx0=v*cos(alfa*pi/180);
    float vy0=v*sin(alfa*pi/180);
```

```
    *px=vx0*t;
    *py=-g*t*t/2+vy0*t;
}

int  main()
{
    float x, y; //posicion objeto
    float  xd=100, yd=50; //posicion diana
    float alfa, v0, t; //angulo, velocidad, tiempo
    int i, j, acierto=0;  //var auxilieares
    char pantalla[NUMFILAS][NUMCOLS]; //matriz a
        representar

    do
    {
        //inicializo asterisco en origen y pantalla
        x=0;
        y=0;
        iniciapantalla(pantalla);
        dibujapantalla(pantalla);

        //jugador introduce datos
        printf("\nIntroduzca velocidad inicial: ");
        scanf("%f",&v0);
        printf("\nIntroduzca angulo de disparo (grados):
             ");
        scanf("%f",&alfa);

        //bucle con calculo de la trayectoria
        for(t=0; x<=xd && x>=0;t=t+0.01)
        {
            posicion(v0, alfa, t, &x, &y);
            i=(int)(yd-y)/yd*NUMFILAS/2+NUMFILAS/2-1;
            j=(int)x/xd*(NUMCOLS-1);

            // impresion de * si esta dentro de pantalla
            if(i>=0&&i<NUMFILAS&&j>=0&&j<NUMCOLS)
            {
                //si se alcanza la diana, acaba el bucle
                    del juego
                if(pantalla[i][j]=='X')
                    acierto=1;

```

```
90                     pantalla[i][j]='*';
91                 }
92             }
93             dibujapantalla(pantalla);
94         }while(acierto==0);
95 
96         printf("\nBingo!!!\n");
97     }
```

En esta ocasión, se ofrece directamente el código íntegro del juego 4.3, que utiliza las funciones auxiliares:

- **`iniciapantalla`** (líneas 8 a 33): recibe por referencia una matriz de caracteres y la rellena con espacios en blanco. Sitúa también el carácter '/' a modo de cañón en la esquina inferior izquierda y una diana compuesta por tres 'X' centrada en su última columna.
- **`dibujapantalla`** (líneas 26 a 39): recibe una matriz por referencia y la muestra en pantalla.
- **`posicion`** (líneas 43 a 52): recibe datos de la velocidad inicial, ángulo y tiempo, y devuelve por referencia la posición del objeto en el instante dado, calculada a partir de las ecuaciones de nuestro modelo. Esta función se apoya en la librería **`math.h`** para calcular el seno y el coseno (líneas 47 y 48). Nótese que estas funciones necesitan su argumento en radianes, lo que se resuelve con la conversión correspondiente en la propia invocación (`alfa` viene en grados).

La función `main` comienza con la definición de una serie de variables cuyo uso aparece comentado en el programa (líneas 56 a 60). De todas ellas, destacan `xd` e `yd`, que determinan la posición de la diana. Dado que esta se representa siempre en la misma posición de la pantalla (última columna y fila intermedia de `pantalla`), estas variables actúan como un factor de escala que permite regular lo cerca o lejos que se encuentra la diana. Dicho de otro modo, la primera columna de `pantalla` se corresponde con `x`=0 y la última con `x`=`xd`. Asimismo, la última fila representa `y`=0 y la fila intermedia `y`=`yd`.

Entre las líneas 62 y 94 se representa el código del bucle principal del programa, un `do/while` que permite al usuario disparar asteriscos hasta que acierte (`acierto`=1). Para ello, se inicializa `pantalla` y se piden datos de velocidad y ángulo (líneas 71 a 74). Con estos parámetros, se utiliza un bucle `for` para rellenar con el carácter '*' las posiciones de `pantalla` por las que haya pasado nuestro asterisco cabreado. Dicho bucle actualiza poco a poco el tiempo, invoca a `posicion` para obtener los valores de `x` e `y` en los que se encuentra el asterisco (línea 79) y lo representa en `pantalla` (siempre y cuando se encuentre *dentro* de la matriz). Cuando la posición del asterisco pasa por una 'X', el juego termina y se imprime en pantalla el mensaje correspondiente (línea 96).

La transformación de las coordenadas `x` e `y` a la fila `i` y a la columna `j` de la matriz `pantalla` merece comentario aparte (líneas 80 y 81). El hecho de que `pantalla` tenga su primera fila "arriba" y la última "abajo" complica la correspondencia lo suficiente como para dedicarle unas líneas al problema.

En primer lugar, las líneas 80 y 81 comienzan el lado derecho de la igualdad con `(int)`. Esto se conoce como *cast* o conversión forzada de tipo. De este modo, se le pide a C que el resultado de la expresión que sigue se interprete como un entero, ya que los índices de filas y columnas deben ser enteros. Si no se hubiera utilizado el *cast*, C habría realizado la conversión de forma autónoma, pues los tipos de dato de las variables a la izquierda y derecha de la igualdad son distintos, lo que habría generado alguna advertencia durante la compilación del programa.

El resto de la expresión realiza un mapeo entre nuestro mundo simulado y lo que se representa por pantalla. Por ejemplo, cuando `x` e `y` son cero, `i`=`NUMFILAS-1` y `j`=0, que es donde hemos ubicado el cañón en `iniciapantalla`. Asimismo, cuando `x`=`xd` e `y`=`yd`, `i`=`NUMFILAS/2-1` y `j`=`NUMCOLS-1`, que es donde se representa la diana. De esta manera, los valores de `x` e `y` son representados dentro de la matriz.

Por último, se muestra una captura de pantalla del juego en la figura 4.3. Como puede verse, una estela de asteriscos muestra la trayectoria seguida por nuestro *angry asterisco* hasta impactar en la diana. El programa confirma el éxito del lanzamiento con la impresión del mensaje "Bingo!".

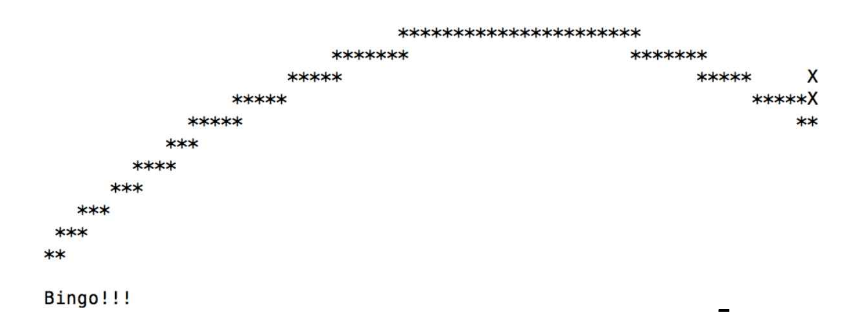



**Figura 4.3:** Pantalla de *Angry Asterisco* (juego 4.3).

## 4.6. Ejercicios propuestos

Os proponemos los siguientes retos de programación relacionados con lo estudiado en este capítulo. En esta ocasión, los retos consisten en la implementación de nuevas funcionalidades en los juegos desarrollados. No dudéis en implementar otras que se os puedan ocurrir. Al fin y al cabo, un programa nunca se termina, simplemente se deja de trabajar en él. Siempre habrá algo que pueda añadirse o mejorarse.



**Ejercicio 4.1: Hundir la Flota 2.0**

La versión que hemos creado de Hundir la Flota (juego 4.2) es susceptible de mejora:

- Realiza los cambios necesarios en el código para que gráficamente se muestre el carácter 'W' cuando tras un disparo se haya dado en el agua, quedado así el carácter '*' reservado a los impactos en barcos del enemigo.
- Crea una función para realizar un ataque aéreo especial que haga explotar varias posiciones del tablero enemigo al azar. Esta función puede ser invocada solo una vez durante el juego con un truco, por ejemplo introduciendo la fila 'Z' y la columna 99.



**Ejercicio 4.2: Angry Asterisco 2.0**

Nuestro peculiar *Angry Asterisco* (juego 4.3) admite diversas mejoras:

- Modifica el programa para que la diana aparezca en una posición aleatoria.
- Incrementa el tamaño de la diana (por ejemplo, para que sea un cuadrado de 'X' u otra forma que resulte atractiva).
- Sería interesante que el programa asigne una puntuación a cada disparo. Una opción sencilla sería relacionarlo con el número de caracteres 'X' destruidos por el asterisco. Asimismo, el valor de cada 'X' derribada podría disminuir con el número de intentos.
- Desarrolla una función que indique si el jugador se ha pasado el juego. El jugador no puede ganar hasta que no haya acabado con todos los caracteres 'X' de la pantalla. ¡Que no quede ni uno!
- Define un número máximo de lanzamientos de asterisco como condición adicional para la finalización del juego. Al acabar, el programa indicará al jugador la puntuación total obtenida.

# Capítulo 5

# Más interacción: lectura y escritura de ficheros

El objetivo primordial del presente capítulo consistirá en aprender una forma útil y sencilla de generar un intercambio de información del que nuestros programas puedan verse beneficiados. Para realizar este intercambio utilizaremos un canal de comunicación que suele denominarse entrada y salida de ficheros, los cuales se convierten en una fuente de datos que se ponen al servicio del programador.

Para afrontar las funciones que vamos a aprender a continuación, os recomendamos que penséis en el capítulo dos; habíamos aprendido cómo establecer comunicación con la máquina utilizando principalmente los comandos `printf()` y `scanf()`. A la hora de ejemplificar tales funciones, hablamos en todo momento de la entrada y salida estándar: el teclado y la pantalla respectivamente. Bien, pues ahora pasamos a presentaros dos funciones muy emparentadas con las que ya conocemos, hasta tal punto que se escriben prácticamente igual: `fprintf()` y `fscanf()`. La diferencia más notoria radica en que las nuevas funciones no interaccionan con entrada y salida estándar, sino que lo hacen mediante la lectura y escritura de ficheros.

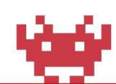
## 5.1. Ficheros... ¿Qué son ficheros, tesoro?

Antes de abordar las vastas posibilidades que nos proporcionan las funciones `fprintf()` y `fscanf()` a la hora de manipular ficheros, conviene que aportemos un poco de contexto al asunto. Comencemos por lo más básico y preguntémonos qué son los ficheros. Básicamente, estamos hablando de agrupaciones de datos destinadas a convivir en dispositivos de gran capacidad, ya sean discos magnéticos, ópticos o incluso cintas. Alguien podría pensar que las cintas son herramientas de almacenamiento arcaicas, y seguramente esté en lo cierto: allá por 1952, el UNIVAC –primer ordenador comercial manufacturado en Estados Unidos– comenzó a utilizar una interfaz denominada Uniservo, capaz de escribir información (que previamente había sido capturada

desde tarjetas perforadas) en cintas magnéticas. Hoy día se siguen utilizando cintas para salvaguardar copias de seguridad de elevado tamaño.

Volviendo a los ficheros, hay que apuntar que el lenguaje C se encuentra plenamente capacitado para explotar y tratar dichos datos a través de diversos procedimientos. Con todo, C impone al programador dos requisitos indispensables para que todo el proceso se lleve a cabo de manera correcta: el primer requisito es abrir el fichero; el segundo, cerrarlo. Parece simple, pero es crucial que la vía de comunicación con el fichero comience y concluya de forma correcta si queremos que nuestros datos estén a salvo y disponibles para otros usuarios. Entre la apertura y el cierre podremos realizar operaciones de manipulación de ficheros –tanto lectura como escritura– a través de las funciones previamente mencionadas.

Los ficheros pueden ser de dos tipos: binarios y de texto. Los ficheros binarios contienen una copia exacta de zonas de la memoria. Así, si un entero ocupa 2 octetos, para almacenar el número entero en un fichero binario basta con copiar esos dos octetos en el soporte del que dispongamos. Por su parte, los ficheros de texto almacenan la información mediante caracteres del código ASCII. De este modo, el contenido del fichero es similar a lo que aparece en la salida que utilicemos, ya sea el monitor o la impresora.



**Ejemplo 5.1: Tipos de ficheros**

A continuación, vamos a ver un ejemplo que mostrará la diferencia entre ambos tipos de ficheros. En primer lugar, un fichero binario, y en segundo término, uno de texto.

**Código**

```
#include <stdio.h>
int main()
{
    int x;
    x = 129;

/* Aqui guardariamos en un fichero el contenido de la
     variable x */
}

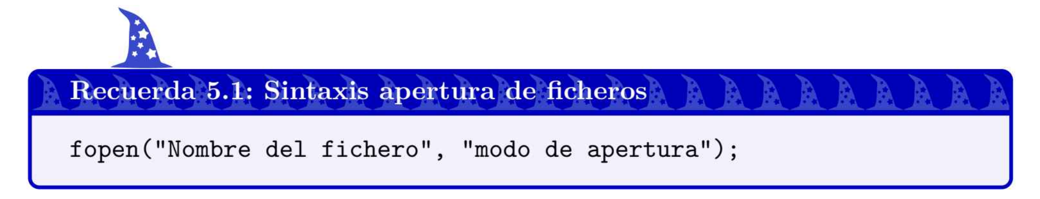
/* Contenido del fichero binario (16 bits) */
0000 0000 1000 0001
/* Contenido del fichero de texto */
0011 0001 0011 0010 0011 1001
49 50 57
```

En este pequeño programa de muestra hemos creado una variable de tipo entero y le hemos asignado el valor numérico 129. Como puede verse, si volcásemos el contenido de la variable en un fichero, el aspecto que tendría cada tipo de fichero sería muy distinto. En el caso del fichero binario sería la representación de la memoria de dicho número, mientras que en el caso del fichero de texto se almacenarían los octetos correspondientes a 49, 50 y 57, que son, a su vez, los códigos ASCII de los números 1, 2 y 9.

Tras esta breve introducción, estamos plenamente capacitados para que nuestros programas dejen huella de forma permanente en nuestro soporte de almacenamiento. Bueno, al menos mientras pervivan los ficheros.

## 5.2. Abrir y cerrar ficheros: el puntero FILE

Lo primero es lo primero; habíamos anunciado en el párrafo anterior de la rigurosidad con que C trata los ficheros. Así pues, vamos a ver cómo podemos abrir y cerrar un fichero.

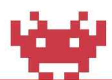


**Recuerda 5.1: Sintaxis apertura de ficheros**

```
fopen("Nombre del fichero", "modo de apertura");
```

La apertura de fichero creará una vía de comunicación entre nuestro programa y el conjunto de datos. Para proceder, el paso inicial será definir un puntero a un tipo de datos denominado FILE, el cual contiene una serie de estructuras internas capaces de mantener en la memoria la información del fichero.

En este punto surge una casuística que será muy habitual a lo largo de nuestra vida como programadores, ya que nos resultará necesario incluir en nuestro código un archivo de librería, que no es otra cosa que un conjunto de tipos de datos y funciones previamente definidos por alguien que, con toda seguridad, tiene más experiencia y sabiduría que nosotros. Así, para hacer uso de FILE, es indispensable que agreguemos mediante la directiva `#include` a una vieja amiga a nuestro programa: la librería de entrada y salida estándar `stdio.h`, puesto que es ahí donde se define el tipo de dato que nos interesa. Por tanto, conviene tener presente que las librerías no solo aportan funciones adicionales; también incluyen tipos de datos nuevos que simplifican el acceso a los recursos de la máquina. Como veremos en los próximos ejemplos, el tipo de dato FILE y las funciones para la interacción con ficheros incluidas en `stdio.h` facilitarán enormemente nuestra labor como programadores. Veamos, en primer lugar, cómo se crea un puntero de tipo FILE.



**Ejemplo 5.2: Creación de puntero a fichero**

Declarar un puntero a fichero es tan simple como teclear una única línea de código, tal y como puede verse en el siguiente ejemplo.

**Código**

```
#include <stdio.h>
main()
{
    FILE * fp; /* Se declara fp como un puntero a
        fichero */
}
```

La función `fopen` recibe dos argumentos:

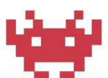
1. El primer argumento es el nombre del fichero, que debe ser válido en el sistema de explotación para evitar posibles errores.

2. El segundo argumento define el modo en el que vamos a trabajar con el fichero tras realizar su apertura. Así, a través de este argumento indicaremos si el fichero será usado para lectura o escritura, el modo de operar –binario o texto– y si se respetará o no el contenido previo del archivo en caso de que existiera.

   Los modos disponibles son:

   - `wt` Crea archivo de texto para escritura, borra contenido previo si lo hubiere.
   - `rt` Abre archivo de texto para lectura.
   - `at` Abre o crea un archivo de texto para escribir al final.
   - `wb` Crea archivo binario para escritura, borra contenido previo si lo hubiere.
   - `rb` Abre archivo binario para lectura.
   - `rb` Abre o crea un archivo binario para escribir al final.

Prosigamos con un ejemplo en el que el programador se comporta como un ser precavido y se adelanta al posible mal funcionamiento del programa. En concreto, se va a prever que la apertura del fichero pueda ser errónea, ya sea porque el soporte de almacenamiento esté lleno o porque el fichero se encuentre protegido contra escritura. En esos casos, la función `fopen()` devuelve 0. Dado que hay programadores que consideran que la inclusión de números en el código fuente es una aberración que limita la comprensión del mismo, es frecuente que se utilice la constante numérica NULL en lugar del 0, aunque su uso es totalmente equivalente. Dicha constante viene definida también, cómo no, dentro de la librería `stdio.h`.



**Ejemplo 5.3: Apertura de fichero**

Prevención de resultado de la invocación de la función fopen() mediante la comprobación del valor obtenido tras intentar abrir el fichero correspondiente.

**Código**

```
#include <stdio.h>
main()
{
    FILE * fp;

    fp = fopen("fichero.dat", "w");
    if (fp == NULL)
    {
        printf("Error de apertura de fichero");
    }
}
```

Tras abrir el fichero y manipularlo, procederemos a cerrar la vía de comunicación que previamente habíamos abierto. Para ello, recurrimos a la función `fclose()` que, como era de esperar, recibe un único argumento: el puntero a FILE con el que hemos abierto el fichero. Una vez más, es práctica recomendada el comprobar el valor retornado por la función para cerciorarnos de que todo fue sobre ruedas y el fichero quedó cerrado (valor de retorno 0) o que, por el contrario, se produjo un error al cerrar (valor de retorno -1).

**Recuerda 5.2: Sintaxis cierre de fichero**

```
int fclose(FILE *);
```

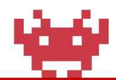
## 5.3. Manipulando ficheros mediante fprintf()

Para comprender las posibilidades que nos ofrece la función `fprintf()` nos será de gran ayuda recordar todo lo que hemos practicado con su función hermana, `printf()`. Básicamente, `fprintf()` escribirá datos utilizando una salida dirigida hacia un fichero en particular, en lugar de la salida estándar. Dichos datos serán pasados a la función `fprintf()` como argumentos. Cada invocación a dicha función retornará el número de bytes escritos en fichero, o un 0 si se produjo un error.

La sintaxis completa es la siguiente:

**`int fprintf(FILE *, cadena de control, lista de argumentos);`**

En la mano del programador queda cómo obtener la información que será escrita en el fichero, ya se trate de datos estáticos (por ejemplo, una cadena de caracteres constante) o del contenido de variables. En el siguiente ejemplo veremos una mezcla de ambos casos, ya que escribiremos un fichero con constantes de texto y con datos que proveerá el propio usuario al interactuar con el programa.



**Ejemplo 5.4: Practicando apertura de ficheros**

Realizar un programa que solicite al usuario su nombre. A continuación, grabar en un fichero dichos datos, antepuestos por la cadena de texto ”Un nuevo jugador ha entrado en el juego, y su nombre es:”.

**Código**

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE * pf;
    char nombre[50];
    pf = fopen("jugadores.txt","at");
    if(pf == NULL)
    {
        printf("no se puede abrir el archivo\n");
        exit(0);  /* definida en stdlib.h, termina el programa */
    }
    printf("\nBienvenido. Introduce tu nombre y comenzarás el juego");
    scanf("%s", nombre);
    fprintf(fp, "\nNuevo jugador: %s", nombre);
    fclose(fp);  /* Esencial: cerrar comunicación con el fichero */
}
```

A la vista del ejemplo anterior podemos deducir los parámetros aceptados por la función de escritura de ficheros. Vamos a desglosarlos:

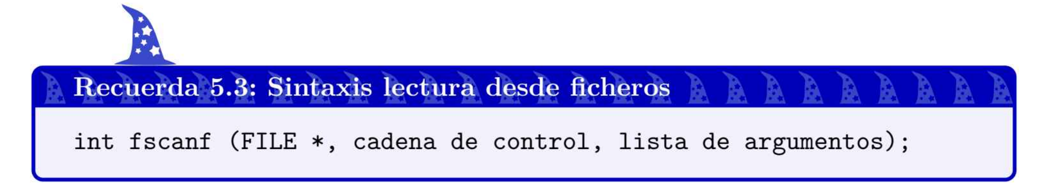
1. El primer argumento es el puntero a FILE, que previamente habrá sido definido para apuntar a nuestro fichero.

2. El segundo argumento es una cadena de control en la que se indican los datos que se van a almacenar en el fichero. En el ejemplo se diferencian cadenas estáticas y especificadores de formato que representan a cada variable que nos interese pasar como argumento, de modo similar a como se operaba con `printf()`.

3. Tras los dos primeros argumentos, se pasará una lista de elementos separados por comas. La lista contendrá tantos argumentos como variables queramos almacenar en el fichero. Tales variables deben haber sido 'anunciadas' en el segundo argumento con los especificadores de formato adecuados. En el ejemplo se pasa la variable nombre, cuyo formato es cadena de caracteres (%s).

## 5.4. Leyendo datos desde ficheros con fscanf()

Una vez que hemos aprendido a almacenar datos en ficheros, examinemos el método que posee el lenguaje C para obtener dichos datos y explotarlos en nuestro programa. Entra en juego, pues, la función `fscanf()`, así que pasemos a examinar su sintaxis:



**Recuerda 5.3: Sintaxis lectura desde ficheros**

```
int fscanf (FILE *, cadena de control, lista de argumentos);
```

De forma similar a su función hermana, cada llamada a `fscanf()` devolverá el número de bytes obtenidos desde fichero, o un 0 si se produjo un error en la lectura. Aunque sus parámetros son similares a los que venimos usando en todo el capítulo, vamos a detallarlos a continuación:

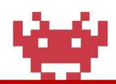
- El primer argumento es el puntero a FILE que apunta al fichero a leer.
- El segundo argumento, la cadena de control, debe concretar el formato de los datos que se desean recuperar desde fichero.
- En tercer lugar, una lista de elementos separados por comas. Cada elemento será una referencia a la dirección de memoria de la variable en la que se almacenará cada dato obtenido desde el fichero. Recordemos que para referenciar variables se utiliza el operador & excepto en el caso de variables de tipo array.

Ahora que ya sabemos explotar un fichero en las dos direcciones, podemos exponer ejemplos más complejos, pero antes debe desarrollarse un poco de contexto histórico previo. A finales de la década de los setenta y principios de los ochenta, los salones recreativos estaban poblados por legiones y legiones de marcianitos. Desde que en 1978 los invasores de *Space Invaders* (Taito) arrasaran desde Japón y *Galaxian* (Namco) consiguiera, un año más tarde, el hito de ser el primer *arcade* a todo color, las compañías desarrolladoras de *arcades* apostaban por explotar este género, conformando

lo que hoy día se conoce como el ilustre panteón de matamarcianos de la historia del videojuego.

Curiosamente, algo tan familiar como almacenar una lista con los nombres de los mejores jugadores que habían participado en la máquina tardó en ser implementado. De hecho, ni *Space Invaders* ni *Galaxian*, mencionados en el párrafo anterior, lucían tal característica. En el año 1980, un juego llamado *Destroyer* aterrizó en los bares españoles, lanzando un difícil reto al jugador: acabar con un monstruo final con forma de demonio cabezón. Si lo conseguían, el juego permitía al héroe grabar su nombre para que se almacenara en la memoria volátil de la máquina, perdurando hasta que se interrumpiese la alimentación. *Destroyer* fue programado por Fernando Yago, un auténtico pionero del *software* de entretenimiento, y manufacturado en los laboratorios de la empresa Electrónica Funcional Operativa (EFO). Muy pocos saben que estamos hablando del primer videojuego español comercial que se conoce.

Nosotros no vamos a aspirar a una hazaña tan elevada –por ahora–, así que nos conformaremos con recuperar aquella lista de valientes jugadores que hemos creado en el ejemplo propuesto en la sección previa del presente capítulo.



**Ejemplo 5.5: Imprimiendo valores desde fichero**

En este ejercicio de ejemplo, el objetivo será construir un programa que sea capaz de imprimir por pantalla la lista de los nombres de los jugadores almacenados en el fichero jugadores.txt

**Código**

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE * pf;
    char nombre[50];
    pf = fopen("jugadores.txt","rt");
    if (pf == NULL)
    {
        printf("No se puede abrir el archivo jugadores.
            txt\n");
        exit(0);
    }
    printf("Lista de mejores jugadores\n");
    while(1)
    {
        fscanf("%s", nombre);
```

```
        if( feof(pf) )
        {
            break;
        }
        printf("%s", nombre);
    }
}
```

Lo más llamativo de este ejemplo es que se utiliza una estructura de control en bucle, `while`, para repetir una y otra vez la lectura de datos desde fichero. Sin embargo, hemos colocado una constante en el lugar en el que habitualmente se ubica una condición explícita de parada. Entonces, ¿cómo podemos asegurarnos de que este `while` no se va a seguir ejecutando hasta el fin de los tiempos? La respuesta está dentro del bucle, donde realizamos una comprobación con `if`: ¿se ha llegado al final del fichero? Si es así, "rompemos" el flujo del código con `break` para salir del bucle `while`.

Pero... ¿cómo podemos saber si se nos ha acabado el fichero? En realidad, cada vez que utilizamos la función `fscanf()`, nos podemos imaginar una cabeza lectora que va avanzando secuencialmente a lo largo del contenido del fichero. Así pues, la comprobación de que dicha cabeza lectora ha alcanzado el final se realiza mediante la función `feof()`, que retornará TRUE en caso afirmativo.

## 5.5. Trivial con ficheros

No tenemos dudas de que el lector recordará con nitidez el ejercicio en el que le proponíamos una sencilla ronda de preguntas y respuestas de cultura general. Sin embargo, el código que se presentó en su momento ofrecía muy poca flexibilidad a la hora de dinamizar el contenido del Trivial en sí mismo, ya que las preguntas y respuestas eran constantes definidas dentro del propio programa.

En 1986, la compañía británica Domark diseñó un magnífico videojuego llamado *Trivial Pursuit Genus Edition* para microordenadores de 8 bits (ver figura 5.1). En este juego podían participar hasta ocho jugadores por turnos, y simulaba a la perfección una partida con el popular juego de mesa.

*Trivial Pursuit Genus Edition* traía un paquete de preguntas por defecto que podía agotarse si la duración de la partida se prolongaba demasiado. Así, los programadores del videojuego optaron por implementar una solución elegante a la par que efectiva: almacenar en la cara B del casete (o en la del disco, dependiendo de la versión) una serie de bloques de preguntas adicionales.

De esta manera, el jugador tenía la posibilidad de acceder al menú de opciones y cargar preguntas nuevas desde un fichero, extendiendo la longevidad del programa sin necesidad de desembolsar más dinero -una estrategia, por cierto, muy distinta a la




**Figura 5.1:** Pantalla de juego de *Trivial Pursuit* de Domark.

que hoy en día está de moda a través de los contenidos descargables, los cuales suelen tener un coste demasiado elevado para lo que ofrecen-.

Por lo tanto, vamos a seguir la estrategia de Domark para mejorar nuestro ejemplo y paliar sus carencias. Para lograrlo, utilizaremos nuestros conocimientos sobre el manejo de ficheros. Partiendo de la base ya conocida, el objetivo será realizar un programa que formule preguntas al jugador, teniendo un fichero denominado "preguntas.txt" como fuente de datos para dichas preguntas.

Para facilitar las cosas y que la solución que proponemos sea lo más estándar posible, adjuntamos a continuación un ejemplo de contenido de fichero de preguntas. Si analizamos el formato en el que se ha incluido dicha información, veremos que cada fila contendrá, de manera sucesiva, el enunciado, las opciones de respuesta y, por último, el número de opción de respuesta que es correcta.

Como en cada fila del fichero existirán varias cadenas de texto, es esencial incluir un carácter que funcione como delimitador, de modo que en el código se pueda distinguir el enunciado de las respuestas. En nuestro ejemplo escogimos el carácter '&'.



**Ejemplo 5.6: Fichero de preguntas**

En el fichero plano que mostramos a continuación, se lista lo que vendría a ser el contenido completo que tendría el fichero de preguntas y respuestas para que sea correctamente leído por el programa principal. Es necesario recordar que el fichero mostrado es un ejemplo, y que los autores del presente libro os invitamos a que experimentéis con más preguntas.

**Código**

```
¿Cuál es el lema de la casa Stark de Invernalia? & Se
    acerca el invierno & Uno para todos y todos para
    uno & Los Stark siempre pagan sus deudas & 1
 Cita el apodo del autor de la novela Don Quijote de
    la Mancha? & El hilarante hidalgo & El manco de
    Lepanto & El potro de Vallecas & 2
¿Con qué comando podemos controlar el flujo de un
    programa en C? & if / else & printf() & scanf() &
    1
```

Y finalmente, el código del juego resuelto:



**Juego 5.1: Trivial con preguntas desde ficheros**

Hacer la competencia a los desarrolladores de Domark y programar el código del juego del Trivial con la condición necesaria de que en el propio flujo del programa se lean las preguntas desde ficheros

**Código**

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    /* Variables de control */
    int opcion;
    int marcador = 0;
    int i;
    int j;

    /* Variables de fichero */
    FILE * pf;
    /* Definición de tipo cadenaCaracteres como array
        de caracteres */
    typedef char cadenaCaracteres [50];
    /* Array que almacenará enunciados de preguntas */
    cadenaCaracteres arrayPreguntas[10];
    /* Matriz que almacenará respuestas posibles para
```

```
        las preguntas */
    cadenaCaracteres matrizRespuestas[10][3];
    /*Array que almacenará la respuesta correcta para
        cada pregunta */
    int arrayCorrectas[10];

    /* Presentación */
    printf("Bienvenido a la segunda versión del juego
        de preguntas y
    respuestas\n");
    printf("¡Ahora, leyendo las preguntas desde disco!\
        n");
    printf("Demuestra tu cultura general y logra un
        gran marcador\n");

    /* Lectura de datos desde fichero */
    pf = fopen("preguntas.txt","rt");
    if(pf == NULL)
    {
        printf("No se puede abrir el archivo\n");
        exit(0);
    }

    i = 0;
    while (1)
    {
        fscanf(pf, "%[^&]|%[^&]|%[^&]|%[^&]&%d",
        &arrayPreguntas[i], &matrizRespuestas[i][0],
        &matrizRespuestas[i][1], &matrizRespuestas[i
            ][2],
        &arrayCorrectas[i]);
        i++;
        if(feof(pf) || i == 10){
            break; /*Romper bucle cuando lleguemos a
                final de fichero
            o si ya hemos leído 10 preguntas */
        }
    }
    fclose(pf);

    /* Bucle que recorre datos previamente leídos y los
        va
    lanzando al jugador */
    j = 0;
```

```
    while (j < i)
    {
        printf("%s \n",arrayPreguntas[j]);
        printf("1 - %s \n",matrizRespuestas[j][0]);
        printf("2 - %s \n",matrizRespuestas[j][1]);
        printf("3 - %s \n",matrizRespuestas[j][2]);
        printf("Elige una opción introduciendo un número (1,2 o 3)");

        scanf("%d",&opcion);

        if (opcion != 1 \&\& opcion != 2 \&\& opcion != 3)
        {
            /* Opción ilegal */
            printf("No has seguido las reglas del juego. No ganas
            ningún punto.\n");
        }
        else
        {
            /* Opción legal. Verificar si es la correcta */
            if (opcion == arrayCorrectas[j])
            {
                /* Acierto. Sumar un punto al marcador */
                printf("Enhorabuena. Respuesta correcta.\n");
                marcador++;
            }
            else
            {
                printf("Respuesta incorrecta. \n");
            }
        }
        j++;
    }

    /* Marcador final */
    printf("Fin de la partida. Tu puntuación final ha sido de %d",
    marcador);
}
```

Y con este código conseguimos un Trivial que resultará muy sencillo de actualizar en lo que a preguntas y respuestas se refiere, sin necesidad de modificar la programación original. Pasamos a comentar varios asuntos acerca de la solución que os hemos propuesto. Lo primero, el comando **`typedef`**, una palabra reservada en lenguaje C cuya misión es definir un tipo de datos y darle un nombre identificativo para su posterior utilización. Se trata de un comando que veremos en profundidad más adelante.



**Recuerda 5.4: Sintaxis apertura de ficheros**

```
/* Definición de tipo cadenaCaracteres como array de caracteres */
typedef char cadenaCaracteres [50];
/* Array cuyos elementos son de tipo cadenaCaracteres */
cadenaCaracteres arrayPreguntas[10];
```

En la solución propuesta nos venía bastante bien utilizar **`typedef`**, puesto que el andamiaje de información requerido para guardar los datos leídos desde fichero incluía *arrays* y matrices de cadenas. Como ya sabemos, una cadena en C se define como un *array* de caracteres, así que es buena práctica que definamos nosotros mismos un tipo cadena de dicha manera. Una vez definido dicho tipo, declarar variables vectores o matrices cuyos elementos sean cadenas se convierte en coser y cantar.

El segundo caballo de batalla de nuestro código se encuentra en los especificadores de formato que se proporcionan en la llamada a la función **`fscanf()`**. La función tiene como objetivo leer líneas de nuestro fichero de preguntas, fichero que definimos previamente como una sucesión de cadenas de caracteres delimitadas por el carácter *ampersand* (**`&`**).



**Recuerda 5.5: Sintaxis apertura de ficheros**

```
fscanf(pf, "%[^|]|%[^|]|%[^|]|%[^|]|%d", &arrayPreguntas[i],
&matrizRespuestas[i][0], &matrizRespuestas[i][1],
&matrizRespuestas[i][2], &arrayCorrectas[i]);
```

Hasta ahora no habíamos utilizado delimitadores explícitos al invocar a **`fscanf()`**, aparte de que siempre echamos mano del especificador **`%s`** para referirnos a una cadena de caracteres. Sin embargo, aquí tendríamos un problema si lo usamos, ya que las cadenas que queremos leer y almacenar en variables incluyen espacios en blanco. Por lo tanto, hemos tenido que utilizar una expresión regular para englobar a los enunciados de preguntas y respuestas: **`[^&]`**. Esta expresión quiere decir: "selecciona todos los caracteres excepto el **`&`**".

Tras incluir dicho modificador, se especifica el caracter delimitador `&`. Así, hasta cuatro veces, ya que tenemos que leer una pregunta y tres opciones de respuesta. Se trata de una buena manera de comprobar que la llamada a `fscanf()` puede complicarse dependiendo de la complejidad de la estructura del fichero en cuestión.

## 5.6. Ejercicios propuestos

Para concluir el capítulo os proponemos diversos retos de programación. Recordad, a programar se aprende programando. ¡Ánimo!



**Ejercicio 5.1: Asciimation**

Aunque cueste creer, en `http://www.asciimation.co.nz` puede encontrarse una versión en caracteres de texto ASCII de *Star Wars*. El proyector reproduce una serie de fotogramas contenidos en un fichero de texto como el que se muestra en la figura 5.2. Como puede verse, el comienzo de cada fotograma se indica con una línea que contiene un asterisco en la primera y última posición. De este modo, el reproductor imprime en pantalla un fotograma, espera una cierta cantidad de tiempo, borra la pantalla y muestra otro fotograma. Crea un reproductor en C para reproducir este tipo de ficheros. (Pista: hay funciones para hacer que un programa de C espere, aunque una alternativa sencilla es introducir un bucle que no haga nada en particular pero que entretenga al ordenador el tiempo suficiente antes de pasar al siguiente fotograma.)



**Ejercicio 5.2: Puntuaciones máximas**

Un clásico del mundo *arcade* son las listas de jugadores con puntuaciones más altas. Desarrolla funciones para añadir este tipo de listas a los juegos anteriormente desarrollados. Para ello, crea un fichero con nombres y puntuaciones cuyo contenido se muestre por pantalla al terminar el juego. Por otro lado, si un jugador ha terminado con una puntuación lo suficientemente elevada, su nombre y su puntuación deben ser convenientemente añadidos al mismo.

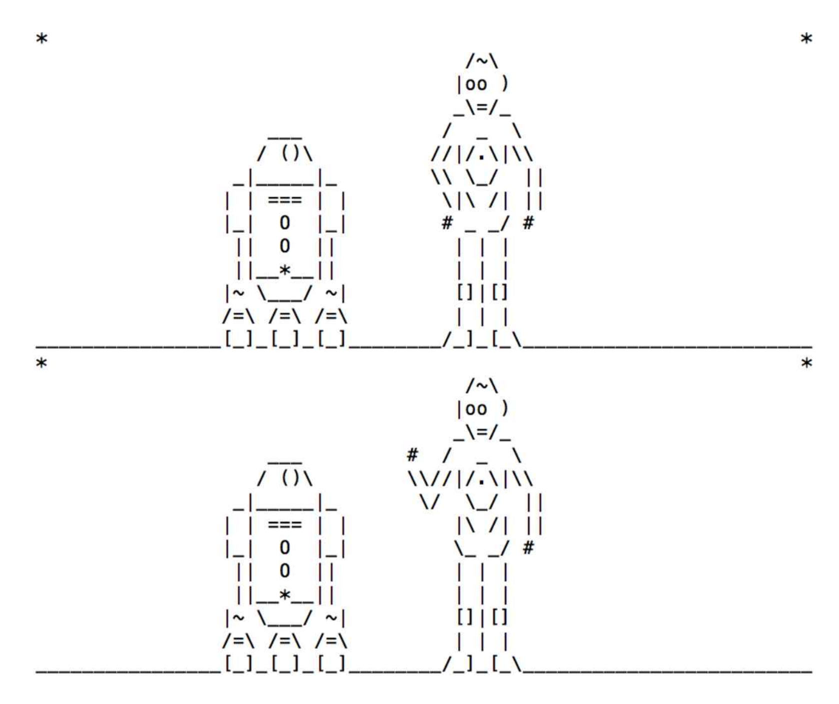


**Figura 5.2:** Dos frames de *Star Wars* en versión ASCII.



### Ejercicio 5.3: Hundir la Flota 3.0

La versión que hemos creado de Hundir la Flota (juego 4.2) ha tenido tanto éxito que se requiere una nueva ampliación para que los usuarios puedan crear un fichero de texto con sus tableros. Para ello, pondrán un 0 donde haya agua y un 1 donde haya un barco. El programa debe pedir a cada usuario el nombre del fichero donde tienen sus barcos, comprobar si es correcto y abrirlo para cargar la información correspondiente en el programa.

# Capítulo 6

# Estructuras simples de datos

Cuando nos adentremos en el maravilloso mundo laboral orientado a la informática, lo más habitual será encontrarnos con problemas reales, y los problemas reales suelen ser, en la mayoría de ocasiones y por desgracia, complejos. Todo problema requiere de una solución, y para hallarla es muy probable que necesitemos modelar entidades y objetos tangibles, convirtiéndolos en datos que podamos manejar a través de nuestro lenguaje de programación favorito.

En el presente capítulo nos atreveremos a dar un paso más en el conocimiento del lenguaje C, utilizando un buen puñado de conceptos básicos ya aprendidos y mezclándolos para crear algo más grande y efectivo que llamaremos *estructura de datos*. La Real Academia Española de la Lengua asegura que una estructura no es más que una disposición o modo de relación entre las distintas partes que forman un conjunto. Dicha definición puede ser válida para hacernos a la idea de lo que tenemos entre manos: un conjunto compuesto de varios datos, cada uno de estos con un nombre y un tipo propios.

Una buena pregunta que debería hacerse el lector sería por qué llamar al capítulo "Estructuras simples de datos", sobre todo teniendo en cuenta que ya hemos dicho que utilizaremos dichas estructuras para afrontar situaciones complejas. Aunque esta pregunta será convenientemente respondida en el próximo capítulo, adelantaremos que, por ahora, vamos a estudiar una estructura de datos básica –o simple– cuyo mero concepto, como todo en esta vida, puede complicarse.

Antes de entrar en materia propiamente y explicar cómo se define una estructura de datos, vamos a plantear una nueva cuestión al lector para comenzar a encauzar el enfoque lúdico del capítulo. Os instamos a pensar en la siguiente situación de hipótesis: imaginad que disponéis de una envidiable capacidad para ser creativos, que vuestra febril imaginación suele desbordarse a menudo y que sentís la necesidad de volcar vuestra inspiración en un papel. Entonces, os sentáis en vuestro escritorio y os apetece crear. Crear un personaje, un escenario, una serie de objetos que encontrar, obstáculos, enemigos, decisiones y un objetivo final. Dicho de otra manera, intentad encarnaros en un escritor de aventuras. ¿Ya? De acuerdo. Ahora, viremos un poco

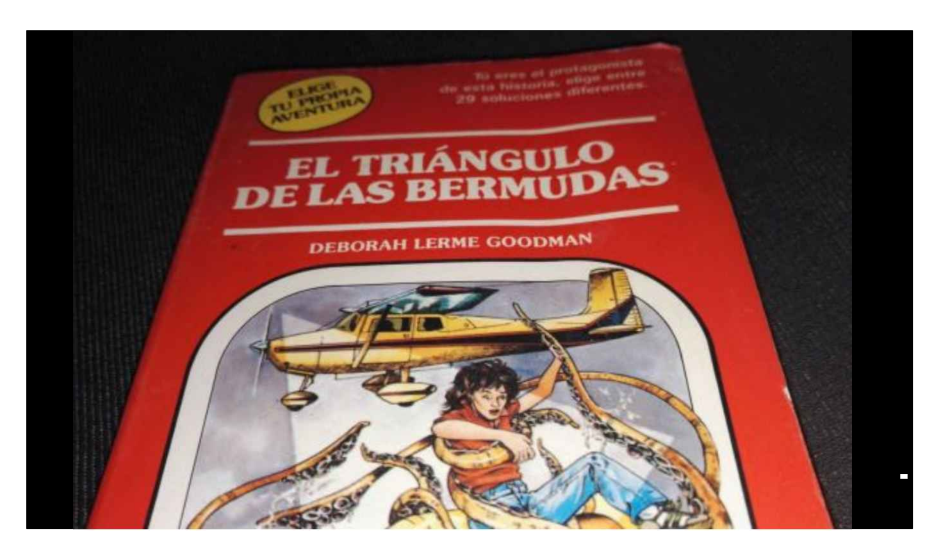



**Figura 6.1:** Ejemplar de colección *Elige tu propia aventura* - Timun Mas.

nuestra hipótesis. Resulta que nosotros, como escritores de aventuras, queremos dotar de interactividad a nuestra creación. Deseamos que el lector se involucre en nuestra historia, tanto, que pueda ser capaz de participar y decidir el curso de la misma.

Al llegar a este punto, muchos de vosotros habréis materializado en vuestros pensamientos ciertos libros que causaron furor en todo el planeta a finales de los años setenta. Fue exactamente en el año 1977 cuando Raymond Almiran Montgomery logró publicar *Journey under the Sea*, la novela pionera de una colección que sería conocida como *Choose your own adventure*. Desde que aquel viaje submarino diera el pistoletazo de salida, se publicarían varias series de libros a raíz de *Journey under the Sea*. En la figura 6.1 se muestra la portada de una de estas obras.

Los tomos que pertenecían a dichas colecciones tenían un factor común, destilando un estilo propio de narración a través del que se depositaba en el lector la responsabilidad de conducir los pasos del protagonista de la historia mediante la toma de decisiones ante un camino que se bifurcaba en dos ramificaciones.

Así, en ciertos puntos de la lectura se instaba al poseedor del libro a realizar un salto hacia una página concreta, a partir de la cual continuaría el argumento. Un ejemplo sería el siguiente: "Si quieres tomar el camino de la izquierda, ve a la página 15. Si por el contrario escoges el de la derecha, continúa en la página 20". Como consecuencia, el punto y final de la historia podría variar, dependiendo de las decisiones tomadas por el propio lector; la naturaleza aventurera de la práctica totalidad de los libros haría que el protagonista pudiera obtener valiosos tesoros, desentrañar enigmas o, simplemente, lograr sobrevivir a una situación de peligro.

Por supuesto, los finales fatales también eran posibles; en numerosas ocasiones podíamos fallar de forma miserable a la hora de conseguir nuestro objetivo. Tras

culminar la historia, era factible reiniciarla desde el principio y escoger decisiones diferentes a las de la primera vez, viviendo una aventura totalmente nueva.

En nuestro país llegamos a conocer estos libros mediante la denominación de *Librojuegos* o *Elige tu propia aventura*, siendo su fascinante concepto la clave de su éxito. Y bien, ¿creéis que sería complicado de trasladar este tipo de libros a un programa de ordenador? Una vez introducida esta cuestión, vamos a dejarla aparcada por el momento. No os preocupéis, la retomaremos más adelante. Ahora, más vale despojarnos de nuestra indumentaria de exploración, dejar la mochila en el suelo y sentarnos a aprender cómo demonios podemos crear una estructura de datos.

## 6.1. La palabra reservada struct

En el lenguaje C, una estructura de datos es una colección de variables que se referencian bajo el mismo nombre, proporcionando un medio conveniente de mantener junta información relacionada. La estructura de datos aporta una gran ventaja para el programador, puesto que le otorga la posibilidad de tratar la información que contiene como un todo. Así pues, la estructura de datos es un conjunto de variables que guardan algún tipo de relación entre sí y que referenciamos mediante un nombre único.

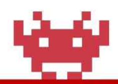
Por ejemplo, si estuviéramos programando una aplicación para nuestra empresa y quisiéramos guardar la información de un empleado, nos interesaría que los siguientes tipos de datos estuvieran relacionados: nombre, número de empleado y edad. Por lo tanto, nuestra solución informática consistiría en crear una estructura de datos `empleado` que internamente pudiera contener como miembros, por un lado, a una variable de tipo *array* de caracteres que almacenasen el nombre, y por otro, a dos variables más de tipo entero que guardarsen su número de empleado y su edad. Para definir todo esto, utilizaremos la palabra reservada `struct` con la siguiente nomenclatura:

**Recuerda 6.1: Sintaxis estructura de datos**

```
struct nombre_estructura
{
   tipo_variable_1 nombre_variable_miembro_1;
   tipo_variable_2 nombre_variable_miembro_2;
   ...
   tipo_variable_n nombre_variable_miembro_n;
}
```

La definición de la estructura puede realizarse en diferentes puntos del código, aunque lo más sencillo es hacerlo tras las directivas `include` y `define`. De este modo se crea dentro del programa un nuevo tipo de dato llamado `nombre_estructura`. Es

decir, a la familia de tipos de dato ya conocidos (`int`, `char`, etc.) se les suma un nuevo miembro. Como vemos en el siguiente ejemplo, la declaración de variables del nuevo tipo se realiza de forma convencional.



**Ejemplo 6.1: Estructura de datos de empleado**

A continuación, vamos a aplicar la sintaxis que hemos presentado al ejemplo del empleado y a definir una variable de dicho tipo de datos llamada `a`.

**Código**

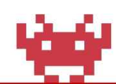
```
#include <stdio.h>

struct empleado
{
    char nombre[30];
    int numero_empleado;
    int edad;
}

main()
{
    struct empleado a;
}
```

Tras codificar dichas líneas, en nuestro programa tendremos una definida estructura llamada *empleado* con la que podremos modelar la información que nos interesa para nuestra empresa. Después de haber definido la estructura, podemos crear tantas variables de dicho tipo como queramos, tal y como se realiza con la variable `a`.

Es factible derribar dos pájaros de un tiro, definiendo la estructura de datos y declarando variables en una sola instrucción. En este ejemplo se realiza la definición del nuevo tipo de dato de forma local a la propia función `main` (a diferencia de lo que se hizo en el ejemplo 6.1).



**Ejemplo 6.2: Definición de variable de estructura de datos empleado**

Como caso práctico que ilustra lo comentado, definamos tres variables (`a`, `b` y `c`) que aprovechen el momento en el que definimos la estructura de datos empleado.

**Código**

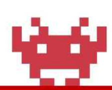
```
#include <stdio.h>
main()
{
    struct empleado
    {
        char nombre[30];
        int numero_empleado;
        int edad;
    } a, b, c; /* Tres variables de tipo empleado: a, b
        y c */
}
```

Una vez que tenemos nuestra estructura definida y nuestra variable de dicho tipo declarada, podremos utilizarla para que almacene datos en su interior. Para conseguirlo utilizaremos el operador `.` que nos permitirá acceder a cada miembro de la estructura de datos por separado. La sintaxis, muy sencilla, es la siguiente:



**Recuerda 6.2: Sintaxis acceso a miembros de estructura**

`variable_tipo_estructura.miembro`



**Ejemplo 6.3: Utilización de operador .**

Consideremos el siguiente ejemplo: ya tenemos definida la estructura `empleado` y acabamos de declarar la variable `a`, así que nuestra siguiente meta será almacenar en dicha variable los datos de un empleado: el señor Domingo Quevedo, trabajador número 128, que acaba de cumplir 36 primaveras. Tendremos que asignar a cada miembro de la variable su valor correspondiente, utilizando lo aprendido en el cuadro Recuerda 6.2.

Para realizar la asignación de la cadena "Domingo Quevedo" al miembro nombre, utilizaremos la función `strcpy(destino, origen)`, que está incluida en la librería estándar de cadenas `string.h`. El cometido de dicha función es copiar la cadena indicada en el segundo parámetro a la que se suministra mediante el primer parámetro. Gracias a esta función, nos ahorramos programar un bucle que haga esto mismo.

**Código**

```
#include <stdio.h>
#include <string.h>

struct empleado
{
        char nombre[30];
        int numero_empleado;
        int edad;
}

main()
{
    struct empleado a;

    strcpy(a.nombre, "Domingo Quevedo");
    a.numero_empleado = 128;
    a.edad = 36;
}
```

Existe una manera alternativa de realizar la asignación de datos a los miembros de nuestra variable tipo estructura, para lo cual en primer lugar debemos definir dicha estructura tal y como hemos visto en ejemplos anteriores, pero sin llegar a declarar ninguna variable en la definición.

Posteriormente, a la hora de hacer la declaración de la variable, aprovecharemos dicha línea para pasar los valores que queremos que tengan sus miembros, separados por comas y entre llaves.

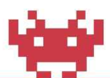
Hay que tener en cuenta que, a la hora de escribir dichos valores, es esencial respetar el orden en el que se definieron cada uno de los miembros de la estructura. En caso contrario, podrán producirse errores de asignación.

A continuación, se incluye la sintaxis completa:

**Recuerda 6.3: Sintaxis asignación en definición de estructura**

```
struct nombre_estructura nombre_variable={valor1, valor2, ... , valorn};
```

Un nuevo ejemplo ayudará a ilustrar la asignación de valores a variables de tipo estructura introducida en el cuadro para recordar 6.3.



**Ejemplo 6.4: Sintaxis de asignación de valores**

Vamos a aplicar la sintaxis de asignación de valores en declaración de variables a nuestro caso práctico anterior, recuperando a nuestro buen empleado Domingo Quevedo, cuya estructura de empleado rellenaremos a partir de los datos pertenecientes a su ficha como trabajador.

**Código**

```
#include <stdio.h>

struct empleado
{
    char nombre[30];
    int numero_empleado;
    int edad;
};

main()
{
    struct empleado a = {"Domingo Quevedo", 128, 36};
}
```

## 6.2. Primer acercamiento al librojuego

Ya sabemos definir estructuras de datos, declarar variables que se aprovechen de dicha estructura y asignarles valores reales. Ahora vamos a encender una hoguera y arrimarnos a su calor con una buena botella de vino, puesto que, por unos instantes, retornaremos al mundo de la aventura interactiva que hemos esbozado al principio del capítulo.

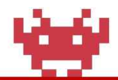
A la hora de computerizar el librojuego que tenemos diseñado en papel, nuestro problema es real y tangible: debemos ser capaces de modelar objetos y trasladarlos al programa mediante lenguaje C, y esos objetos son, obviamente, las páginas de nuestro librojuego. Por lo tanto, idear la forma de representar una página a través de nuestras líneas de código sería una forma magnífica de empezar nuestra aventura, equiparable a encontrar un mapa de un tesoro escondido.

Pero antes de machacar teclas a lo loco, siempre conviene analizar el problema. Estudiemos el objeto en cuestión: una página. ¿Qué es lo que conforma una hoja de un libro interactivo? A bote pronto, la página viene identificada por un número; luego tendríamos una cantidad de texto que se encargará de describir la situación; tras el texto vendría la pregunta de rigor, esa pregunta que dibuja ante el lector una

bifurcación de caminos, y ante la cual debe decidir hacia donde encaminarse. Además, la página contendrá las opciones de la pregunta, y cada opción debe llevarnos hacia una página diferente. Por último, debemos determinar si una página simboliza el fin de la aventura. Entonces, recapitulemos:

- Número de página.
- Texto de la página.
- Pregunta para el lector.
- Opción de bifurcación 1.
- Opción de bifurcación 2.
- Página destino para la opción de bifurcación 1.
- Página destino para la opción de bifurcación 2.
- ¿Es una página final?

Con estas premisas, procedamos a implementar mediante `struct` la página estándar.



**Ejemplo 6.5: Primer acercamiento al librojuego computerizado**

En este ejemplo vamos a codificar la estructura de nuestra página estándar, incluyendo en la misma todos y cada uno de los campos que previamente hemos analizado. Después de definir la estructura de página, crearemos una variable de dicha estructura, inicializándola con valores válidos para los campos definidos.

**Código**

```
#include <stdio.h>

struct pagina
{
    int numero;
    char texto[300];
    char pregunta[100];
    char opcion1[100];
    char opcion2[100];
    int pagina_destino1;
    int pagina_destino2;
    int es_final; /** Valor 0 = NO, valor 1 = Sí **/
```

```
};

main()
{
    struct pagina inicio = {1, "Estás explorando una
        oscura caverna a la luz de tu antorcha. De
        repente, te sientas a descansar en un  húmedo
        risco que sobresale del suelo. Bajas la vista y
         ante tus ojos se materializa un brillante
        anillo", "¿Qué decides hacer con dicho objeto?"
        , "Lo coges y te lo colocas en el dedo anular",
         "Ignoras el anillo", 2, 3, 0};
}
```

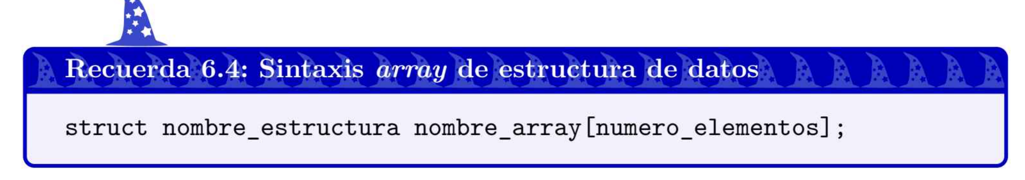
Una vez finalizada nuestra labor de definición de lo que sería una página estándar, habremos avanzado un buen trecho, pero aún nos queda averiguar cuál sería la forma más óptima de definir todas y cada una de las páginas que contendrá nuestro librojuego.

Sin duda, queda mucha aventura por recorrer. Sigamos avanzando por el capítulo. Como consejo, será mejor que dejes la hoguera encendida mientras tanto.

## 6.3. *Arrays* de estructuras de datos

La manera más práctica de utilizar las estructuras de datos se basa en el uso de *arrays* o vectores. Como ya sabemos, en un *array* podemos disponer de una sucesión de variables ordenadas de manera secuencial, cada una de estas variables referenciadas por un índice numérico. Por tanto, parece razonable que queramos definir un *array* de estructuras de datos para sacar el máximo partido al potencial de procesamiento de datos que posee la palabra reservada `struct`.

La sintaxis para definir un *array* de estructuras es la habitual una vez que se ha definido la estructura, ya que esta se convierte en un nuevo tipo de dato:

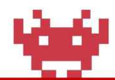



Si nos fijamos en la forma de acceder a cada elemento del *array*, resultará idéntica a la que ya hemos utilizado en temas anteriores, utilizando el operador `[]` para tener acceso al elemento que deseemos, indicando el índice numérico en cuestión, seguido del operador `.` y el nombre del miembro de la estructura que nos interese.



**Recuerda 6.5: Sintaxis acceso a miembros de *array* de estructuras**

`nombre_array[indice_elemento].nombre_variable_miembro_n`



**Ejemplo 6.6: Arrays de estructuras de datos**

Recuperemos de nuevo nuestra estructura de datos Empleado y apliquemos al programa de nuestra empresa la filosofía de utilizar *arrays*. A menos que seamos una empresa muy pequeña, lo más lógico es que tengamos que almacenar varios empleados, los cuales conforman nuestra plantilla. Por tanto, nos será de gran ayuda tenerlos todos agrupados en un vector, que se declararía de la siguiente manera.

**Código**

```
#include <stdio.h>
#include <string.h>
main()
{
    /* Estructura de dato para empleado */
    struct empleado
    {
        char nombre[30];
        int numero_empleado;
        int edad;
    } plantilla[10];

    /* Variables del programa */
    int contador = 0;
    int i;

    /* Asignamos valores a nuestro empleado número 1 */
    strcpy(plantilla[0].nombre, "Domingo Quevedo");
    plantilla[0].numero_empleado = 1;
    plantilla[0].edad = 36;
    contador++;

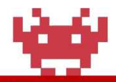
    /* Asignamos valores a nuestro empleado número 2 */
    strcpy(plantilla[1].nombre, "Marta Cornejo");
    plantilla[1].numero_empleado = 2;
    plantilla[1].edad = 40;
```

```
    contador++;

    /* Imprimimos nuestra plantilla de empleados */
    printf("\n Plantilla de empleados");
    for (i = 0; i < contador; i++)
    {
       printf("\n Empleado %i con nombre %s y edad %i",
           plantilla[i].numero_empleado, plantilla[i].
          nombre, plantilla[i].edad);
    }
}
```

Como vemos en la resolución del ejemplo, debemos tener cuidado con la longitud del *array*. En primer lugar, tenemos que dimensionarlo, esto es, asignar un número máximo de elementos. En este caso hemos fijado en 10. Posteriormente, utilizamos una variable auxiliar `contador` que nos ayudará a saber cuántos empleados hemos agregado al *array*. En último lugar, utilizamos la instrucción de control de flujo `for` para recorrer el vector, fijando como condición de parada que se alcance el número de elementos señalados por nuestra variable `contador`.

Por otro lado, el ejemplo 6.6 muestra también que es posible introducir la definición de la estructura dentro de la función `main`. Este uso es particularmente interesante cuando se desea restringir la estructura al ámbito de una determinada función. No obstante, si se desea que la estructura sea accesible desde cualquier función, lo más recomendable es proceder como en los ejemplos precedentes.



**Ejemplo 6.7: Asignación de valores a estructura de datos**

Imaginemos que tenemos una biblioteca de videojuegos enorme, en la que nos surge la necesidad de catalogar todos y cada uno de los ejemplares que almacenamos en la estantería.

Como siempre, lo primero será definir la estructura de datos necesaria para modelar los datos de un videojuego. Estaría bien poder guardar el título, así como el sistema de entretenimiento para el que fue diseñado y, por qué no, una calificación numérica que nos permita valorar de cero a diez cuánto nos gusta ese juego.

Finalmente, procederemos a implementar un código que nos permita preguntar al usuario por los datos de cada videojuego, de forma que sea él mismo el que los introduzca mediante la entrada estándar.

**Código**

```
#include <stdio.h>
#include <string.h>
main()
{
    struct juego
    {
        char titulo[30];
        char sistema[30];
        int nota;
    } biblioteca[10];

    int contador = 0;

    printf("\n Bienvenido a la biblioteca de juegos. Vamos a almacenar los datos de cada uno de ellos");
    while (contador < 2)
    {
        printf("\n Juego %i", contador + 1);
        printf("\n Titulo? ");
        gets(biblioteca[contador].titulo);
        printf("\n Sistema? ");
        gets(biblioteca[contador].sistema);
        printf("\n Nota de 0 a 10? ");
        scanf("%i", &biblioteca[contador].nota);
        contador++;
    }

    /** Imprimimos nuestra biblioteca de juegos **/
    int i;
    printf("\n Biblioteca de juegos");
    for (i = 0; i < contador; i++)
        printf("\n %i: %s (%s). Nota: %i", contador, biblioteca[i].titulo, biblioteca[i].sistema, biblioteca[i].nota);
}
```

Cabe reseñar que en el ejemplo resuelto se combina la utilización de las funciones `scanf()` y `gets()` para recepcionar datos desde la entrada estándar. Mientras que la primera se escoge para los tipos numéricos –la nota que otorgamos a cada juego–, la segunda nos permite tratar las cadenas de caracteres, ya que nos da mucha más flexibilidad, tal y como vimos en el capítulo de entrada y salida de datos. Sin embargo,

de vez en cuando comprobamos que las mezclas suelen conllevar ciertos peligros, y este ejemplo da muestra de ello.

Cuando utilizamos `scanf()` en la línea 23 del código de ejemplo para asignar el valor numérico introducido, el búfer de entrada vuelve a jugarnos una mala pasada, guardándose un carácter `\n` de retorno de carro que es recepcionado por la siguiente llamada a la función `gets()` en la nueva iteración del bucle `for`. Esto provoca que no podamos introducir el título del juego, ya que `gets()` ha interpretado ese retorno de carro como una línea en blanco que asignar al miembro `titulo` de la estructura de datos. Se trata de un problema muy común en lenguaje C, derivado de las combinaciones que suelen aplicarse para tratar la entrada estándar. Sin embargo, la solución no es nada compleja, pues basta con usar la función `getc()`, el cual, recordemos, se encarga de recepcionar un carácter desde la entrada especificada en su principal argumento. Por tanto, basta invocar a dicha función pasando como parámetro la entrada estándar `stdin`.



**Ejemplo 6.8: Corrigiendo nuestra asignación de valores**

Para consolidar nuestros conocimientos, en este ejemplo tomaremos el código del ejemplo anterior, modificándolo y actualizándolo con las funciones apropiadas de recepción de cadenas de texto y caracteres, tal y como hemos aprendido.

**Código**

```
#include <stdio.h>
#include <string.h>
main()
{
    struct juego
    {
        char titulo[30];
        char sistema[30];
        int nota;
    } biblioteca[10];

    int contador = 0;

    printf("\n Bienvenido a la biblioteca de juegos. Vamos a almacenar los datos de cada uno de ellos");
    while (contador < 2)
    {
        printf("\n Juego %i", contador + 1);
        printf("\n Titulo? ");
```

```
        gets(biblioteca[contador].titulo);
        printf("\n Sistema? ");
        gets(biblioteca[contador].sistema);
        printf("\n Nota de 0 a 10? ");
        scanf("%i", &biblioteca[contador].nota);
        contador++;
        getc(stdin); /* Engullir '\n' */
    }

    /** Imprimimos nuestra biblioteca de juegos **/
    int i;
    printf("\n Biblioteca de juegos");
    for (i = 0; i < contador; i++)
        printf("\n %i: %s (%s). Nota: %i", contador,
           biblioteca[i].titulo, biblioteca[i].sistema,
            biblioteca[i].nota);
}
```

## 6.4. Un librojuego solo para aventureros

Bienvenido de nuevo, aventurero. Consideramos que has adquirido la suficiente experiencia como para programar tu primer librojuego mediante lenguaje C. Antes que nada, te recomendamos que dibujes en un papel las páginas de que va a constar tu aventura, así como el entramado de bifurcaciones que poseerá. Utiliza como punto de partida la página de ejemplo que ya fijamos previamente, en la que el protagonista encuentra un anillo y debe decidir qué hacer con él.

En la sección anterior definimos un primer esbozo de lo que sería la estructura de datos de una página del librojuego. Por lo tanto, en este ejercicio parece razonable utilizar un *array* de estructuras, que debemos inicializar con las páginas que contendrá el libro. Además, vamos a aprovechar que el vector nos permite tener "etiquetadas" a las páginas mediante su índice para eliminar el miembro "Número de página" de la estructura, de forma que cuando nos queramos referir a una página en concreto para fijar el destino de una u otra bifurcación, debemos hacerlo indicando el índice que posee dentro del *array*.

Hecho esto, el siguiente paso será utilizar el control de flujo del programa para recorrer ese *array* de forma que se le presente al usuario por pantalla el contenido de la página actual, y se le permita escoger una de las bifurcaciones contenidas en dicha página. Por último, habrá que incorporar a ese control de flujo una condición para determinar si el usuario ha llegado a una página que signifique el final de su aventura. ¡Y eso es todo! Quizás, *a priori*, no parezca una tarea fácil, pero... ¿acaso existe alguna aventura que lo sea?



**Juego 6.1: Un librojuego funcional**

Codifica el librojuego según las premisas explicadas anteriormente, de manera que al ejecutar el código resultante, el programa permita jugar al usuario mientras va leyendo y navegando por las distintas páginas del libro.

**Código**

```
#include <stdio.h>
#include <string.h>
main()
{
    /** Array de estructuras con 6 elementos **/
    struct pagina
    {
        char texto[300];
        char pregunta[100];
        char opcion1[100];
        char opcion2[100];
        int pagina_destino1;
        int pagina_destino2;
        int es_final; /** Valor 0 = NO, valor 1 = Sí **/
    }libro [6];

    /** Declaración de las 6 páginas **/
    libro[0].numero = 1;
    strcpy(libro[0].texto, "Estas explorando una oscura
        caverna a la luz de tu antorcha. De repente,
        te sientas a descansar en un humedo risco que
        sobresale del suelo. Bajas la vista y ante tus
        ojos se materializa un brillante anillo");
    strcpy(libro[0].pregunta, "Que decides hacer con
        dicho objeto?");
    strcpy(libro[0].opcion1, "Lo coges y te lo colocas
        en el dedo anular");
    strcpy(libro[0].opcion2, "Ignoras el anillo y
        prosigues tu camino");
    libro[0].pagina_destino1 = 1;
    libro[0].pagina_destino2 = 3;
    libro[0].es_final = 0;

    libro[1].numero = 2;
    strcpy(libro[1].texto, "Sientes un extraño
        cosquilleo. Cuando empiezas a pensar que no ha
```

```
        sido buena idea colocarte el anillo , percibes
        que una criatura de piel viscosa y ojos
        brillantes se planta ante ti. Grita desesperada
         que ha perdido el anillo");
    strcpy(libro[1].pregunta,"Que decides hacer?");
    strcpy(libro[1].opcion1, "Te quitas el anillo y se
        lo devuelves");
    strcpy(libro[1].opcion2, "La ignoras y continuas tu
         camino");
    libro[1].pagina_destino1 = 2;
    libro[1].pagina_destino2 = 3;
    libro[1].es_final = 0;

    libro[2].numero = 3;
    strcpy(libro[2].texto, "Al quitarte el anillo, la
        criatura se asusta y reacciona abalanzandose
        contra ti. No te esperabas tal movimiento, asi
        que tropiezas y caes, golpeandote en la cabeza
        con el risco. Tu ultima vision en vida es la
        criatura dando saltos de  alegria");
    libro[2].es_final = 1;

    libro[3].numero = 4;
    strcpy(libro[3].texto, "Dos puertas se abren ante
        tus ojos. La de la izquierda refulge en la
        oscuridad con un destello dorado. La de la
        derecha es una verja oxidada y protege las
        tinieblas.");
    strcpy(libro[3].pregunta, "Que puerta abres?");
    strcpy(libro[3].opcion1, "La puerta dorada de la
        izquierda");
    strcpy(libro[3].opcion2, "La verja oxidada de la
        derecha");
    libro[3].pagina_destino1 = 4;
    libro[3].pagina_destino2 = 5;
    libro[3].es_final = 0;

    libro[4].numero = 5;
    strcpy(libro[4].texto,"Nada mas abrir la refulgente
         puerta, ves un pozo profundo abierto en el
        terreno. Te asomas y observas el mismo brillo
        dorado en el fondo. Te atrae tanto que caes sin
         remision por el agujero, quedando atrapado
        para siempre en la cámara del tesoro");
```

```
    libro[4].es_final = 1;

    libro[5].numero = 6;
    strcpy(libro[5].texto,"Tras recorrer una senda
        oscura como la noche, finalmente divisas un
        claro de luz que anuncia que has logrado
        escapar de la caverna. Tus bolsillos siguen
        vacios de tesoros, pero has comprendido que el
        mayor de los tesoros es conservar tu propia
        vida");
    libro[5].es_final = 1;

    /** Estructura de flujo del programa **/
    while (libro[pagina_actual].es_final != 1)
    {
        puts(libro[pagina_actual].texto);
        puts(libro[pagina_actual].pregunta);
        puts("Opcion 1: ");
        puts(libro[pagina_actual].opcion1);
        puts("Opcion 2: ");
        puts(libro[pagina_actual].opcion2);
        scanf("%i", &opcion_escogida);

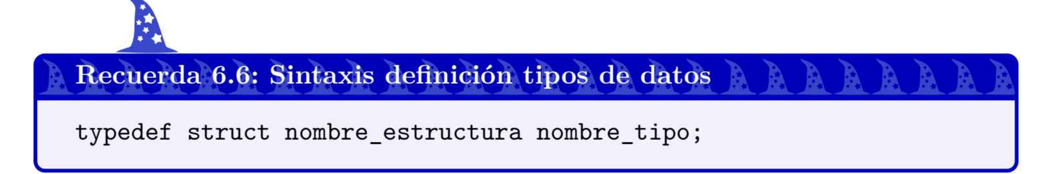
        if (opcion_escogida == 1)
            pagina_actual = libro[pagina_actual].
                pagina_destino1;
        else if (opcion_escogida == 2)
            pagina_actual = libro[pagina_actual].
                pagina_destino2;
    }

    /** Fin: mostrar texto de pagina final **/
    puts(libro[pagina_actual].texto);
    puts("FIN");
}
```

## 6.5. Definiendo tipos sinónimos con typedef

Una vez que hemos comprobado la potencia que alberga una estructura para gestionar un andamiaje de datos de forma ordenada, vamos a examinar la sintaxis de la instrucción `typedef`, la cual nos permitirá definir un tipo de dato sinónimo para el nombre de nuestra estructura, de forma que podamos utilizarlo posteriormente para

evitar escribir siempre **`struct nombre_estructura`** cada vez que hagamos referencia a la misma.



**Recuerda 6.6: Sintaxis definición tipos de datos**

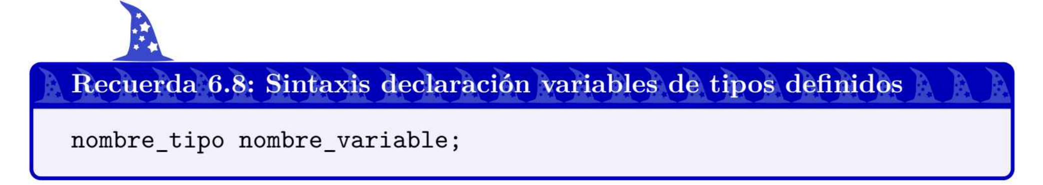
```
typedef struct nombre_estructura nombre_tipo;
```

Con la sentencia del cuadro para recordar 6.6 se crea el nuevo tipo de dato **`nombre_tipo`**, cuyo uso es análogo al de cualquier otro tipo de dato del programa. Dicha definición también puede realizarse junto con la de la estructura, como se indica a continuación.

**Recuerda 6.7: Sintaxis definición tipos de datos**

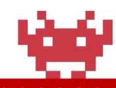
```
typedef struct nombre_estructura
{
   tipo_variable_1 nombre_variable_miembro_1;
   tipo_variable_2 nombre_variable_miembro_2;
   ...
   tipo_variable_n nombre_variable_miembro_n;
} nombre_tipo;
```

Una vez definido nuestro nombre personalizado de tipo de datos de la manera introducida en el cuadro 6.6 o 6.7, podremos declarar variables de dicho tipo de la manera habitual, mediante la sintaxis que se indica en el cuadro siguiente.



**Recuerda 6.8: Sintaxis declaración variables de tipos definidos**

```
nombre_tipo nombre_variable;
```



**Ejemplo 6.9: Un nuevo tipo definido para la biblioteca de videojuegos**

Recuperando el ejemplo de la biblioteca de videojuegos, podríamos definir nuestra estructura como un tipo propio denominado `tipojuego`, para luego declarar uno o varios *arrays* cuyos elementos sean del nuevo tipo.

**Código**

```
#include <stdio.h>
#include <string.h>

typedef struct juego
{
    char titulo[30];
    char sistema[30];
    int nota;
} tipojuego;

main()
{
    tipojuego mi_biblioteca[10];
    tipojuego otra_biblioteca[10];

    strcpy(mi_biblioteca[0].titulo, "Mi juego");
    strcpy(mi_biblioteca[0].sistema, "Mi consola");
    mi_biblioteca[0].nota = 7;

    strcpy(otra_biblioteca[0].titulo,"Juego de otro");
    strcpy(otra_biblioteca[0].sistema, "Otra consola");
    otra_biblioteca[0].nota = 5;
}
```

## 6.6. Estructuras de datos anidadas

Una de las ventajas que nos concede el poder definir un tipo de datos de estructura mediante **`typedef`** consiste en la posibilidad de anidar estructuras de datos dentro de otras, de forma que el código quede claro y legible. La sintaxis sería la siguiente:

**Recuerda 6.9: Sintaxis estructuras anidadas**

```
typedef struct nombre_estructura_anidada
{
   tipo_variable_1 nombre_variable_miembro_1;
   tipo_variable_2 nombre_variable_miembro_2;
   ...
```

```
   tipo_variable_n nombre_variable_miembro_n;
} nombre_tipo_anidado;

typedef struct nombre_estructura
{
   nombre_tipo_anidado nombre_variable_miembro_1;
   tipo_variable_2 nombre_variable_miembro_2;
   ...
   tipo_variable_n nombre_variable_miembro_n;
} nombre_tipo_principal;
```



**Ejemplo 6.10: Una biblioteca con juegos anidados**

Vamos a crear un tipo de estructura de datos llamado `tipobiblio` que almacene los datos de una biblioteca de juegos. Dicho tipo tendrá como miembro principal a un *array* de estructuras tipo `tipojuego`. Para acceder a los miembros de las estructuras anidadas, seguiremos utilizando el operador '.' tal y como hemos venido haciendo hasta ahora.

**Código**

```
#include <stdio.h>
#include <string.h>

typedef struct juego
{
    char titulo[30];
    char sistema[30];
    int nota;
} tipojuego;

typedef struct biblio
{
    tipojuego juego[30];
} tipobiblio;

main()
{
    tipobiblio mi_biblioteca;

    strcpy(mi_biblioteca.juego[0].titulo, "Street
        Fighter II");
```

```
    strcpy(mi_biblioteca.juego[0].sistema, "Super
        Nintendo");
    mi_biblioteca.juego[0].nota = 9;

    strcpy(mi_biblioteca.juego[1].titulo, "Sonic");
    strcpy(mi_biblioteca.juego[1].sistema, "Megadrive")
        ;
    mi_biblioteca.juego[1].nota = 8;
}
```

## 6.7. Ejercicio propuesto

Finalmente, os proponemos el siguiente reto de programación. Para unos aventureros avezados como vosotros, seguro que os resultará pan comido.



**Ejercicio 6.1: Monstruos de bolsillo**

En 1996 Nintendo lanzó el videojuego *Pocket Monsters*, más conocido como *Pokemon*, inaugurando así una de las sagas más exitosas de la historia del videojuego. En esta franquicia el protagonista cuenta con la asistencia de la *Pokedex*, una especie de enciclopedia virtual portátil con datos acerca de estas criaturas. Crea un programa que sirva de base para el desarrollo de una *Pokedex*. Para ello, permitirá al usuario introducir datos de diferentes Pokemon (nombre, tipo, ataques conocidos...) y mostrará dicha información por pantalla. De esta manera, hacerse con todos los Pokemon será un poco más sencillo.

# Capítulo 7

# Estructuras de datos dinámicas

En el capítulo anterior hemos estudiado cómo definir una estructura de datos, inicializándola con valores y utilizándola para cubrir las necesidades de nuestra aplicación. Cierto es que prometimos explicar qué eran las estructuras de datos complejas, así que ha llegado el momento de cumplir nuestra promesa.

Una estructura de datos *compleja* o *dinámica* tiene una ventaja muy significativa respecto a la simple: su tamaño puede variar en el tiempo. De esta manera, la flexibilidad que ofrece es enorme, ya que puede adaptarse a requerimientos más difíciles de cubrir. Sin embargo, como dijo un hombre sabio, todo poder conlleva una responsabilidad.

Y es que definir una estructura de datos dinámica nos exigirá más esfuerzo que el que invertimos en construir las estructuras analizadas hasta el momento, puesto que exhiben un factor común indispensable: deben poseer un campo miembro definido como tipo puntero. ¿Qué supone esta característica? pues el hecho de tener que declarar el esqueleto de cada elemento de la estructura, sabiendo que dichos elementos estarán enlazados unos con otros. Dichos elementos se denominan nodos, y su definición sigue esta sintaxis:

**Recuerda 7.1: Sintaxis estructura de datos dinámica**

```
struct nombre_nodo_estructura_dinamica
{
   struct nombre_nodo_estructura_dinamica *nombre_variable_puntero;
   tipo_variable_2 nombre_variable_miembro_2;
   tipo_variable_3 nombre_variable_miembro_3;
   ...
   tipo_variable_n nombre_variable_miembro_n;
}
```

Como vemos en dicho esqueleto, el primer campo miembro que se debe definir es un puntero que apuntará a otro nodo cuyo tipo será el mismo que estamos definiendo en ese momento. La explicación teórica puede sonar confusa y farragosa, pero como suele pasar, los ejemplos serán los que sean capaces de arrojar luz sobre el asunto. En el presente tema aprenderemos uno de los modelos de estructuras dinámicas de datos más populares dentro de la programación estructurada: las pilas.

## 7.1. Cómo fabricar pilas y no morir en el intento

Una pila es un tipo de estructura en forma de lista de elementos en la que solo se pueden insertar y eliminar nodos desde uno de los extremos de la lista. Tales operaciones se conocen como *push (empujar)* y *pop (tirar)*, de forma respectiva. Además, las escrituras de datos siempre equivalen a inserciones de nodos, mientras que las lecturas se traducen en examinar un nodo. Se trata de un comportamiento LIFO (*Last In First Out*), que traducido resulta que el último elemento en entrar es el primero en salir.

Para entender mejor cómo se opera con una pila, lo más efectivo suele ser trasladarse por unos momentos a una cocina y echar un vistazo a nuestro fregadero. Si miramos nuestra torre de platos por fregar, comprobaremos que solo podemos limpiar el plato que está en la parte superior de dicha torre, mientras que si alguien termina de comer y añade un elemento más, lo hará colocándolo arriba del resto de platos. Al fin y al cabo, esta metáfora nos vale para hacernos un dibujo mental de una pila, que no es más que una torre de elementos.

Vamos a definir lo que sería el nodo más básico de una pila, el cual tendrá un miembro con un tipo de datos base, por ejemplo, un dato de tipo entero, y por supuesto, un miembro puntero que señale al siguiente nodo del mismo tipo de estructura.

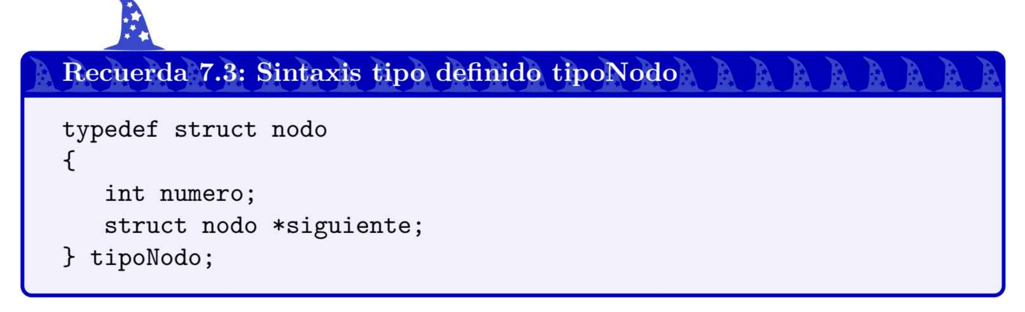
**Recuerda 7.2: Sintaxis nodo de pila**

```
struct nodo
{
   int numero;
   struct nodo *siguiente;
}
```

Una vez que hemos comprendido cómo se define un nodo, vamos a definir tipos concretos que nos ayudarán sobremanera a poder hacer operaciones con las pilas. Para ello echaremos mano de la función `typedef`, que a todos nos debe sonar ya desde hace tiempo. Así, el primer tipo que vamos a definir es `tipoNodo`, cuyo nombre se explica por sí mismo.



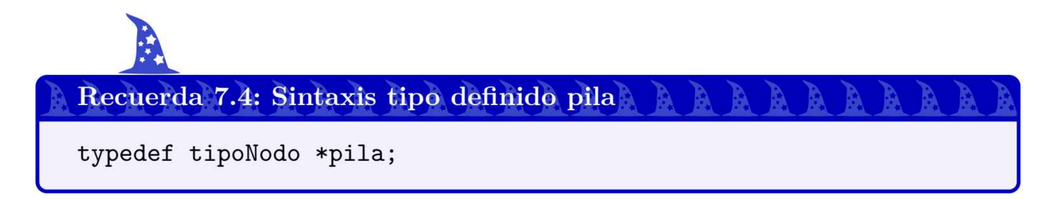

**Figura 7.1:** Representación gráfica de la estructura de una pila.



**Recuerda 7.3: Sintaxis tipo definido tipoNodo**

```
typedef struct nodo
{
   int numero;
   struct nodo *siguiente;
} tipoNodo;
```

Una vez que tenemos el tipo de nodo definido, lo siguiente que se debe definir será el tipo pila propiamente. Si una pila es una relación de nodos, y cada nodo puede apuntar al siguiente, entonces para definir una pila basta con tener un puntero apuntado a un nodo que, como es lógico, se convertirá en el elemento que se ubica en la parte superior de la torre.

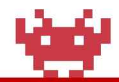


**Recuerda 7.4: Sintaxis tipo definido pila**

```
typedef tipoNodo *pila;
```

## 7.2. *Push*, *Pop* y reservas de memoria

Adentrémonos en la definición de las distintas operaciones que podremos efectuar en nuestra pila. Como se comentó previamente, nuestras posibilidades se reducirán a `push()` y `pop()`.



**Ejemplo 7.1: Haciendo push para introducir elementos**

Realiza una función denominada `push()` que reciba un puntero a un tipo `Pila` y un entero con el fin de crear un nodo en la parte superior de la pila a cuyo miembro `numero` se le asigne el valor recibido.

**Código**

```
void push(Pila *pila, int valor)
{
    tipoNodo *nuevo;

    /* Crear un nodo nuevo */
    nuevo = (tipoNodo *)malloc(sizeof(tipoNodo));
    nuevo->numero = valor;

    /* El siguiente nodo al nuevo es el que antes era
       el tope de la pila */
    nuevo->siguiente = *pila;
    /* Ahora, nuestra pila apuntará al nuevo nodo */
    *pila = nuevo;
}
```

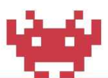
A la vista del código, notaremos que se utiliza la función `malloc()` –la cual se encuentra en la librería `stdlib.h`– a la hora de crear un nuevo puntero a `tipoNodo`.

**Recuerda 7.5: Sintaxis reserva de memoria con malloc()**

```
nuevo_nodo = (tipoNodo *) malloc(sizeof(tipoNodo));
```

Dicha función `malloc()` se utiliza para reservar memoria. Sus siglas provienen del término inglés '**m**emory **alloc**ate" (asignar memoria) y su cometido es tomar un trozo de memoria y devolvernos su dirección, de forma que podamos utilizarla para alojar, en este caso, nuestro nuevo `tipoNodo`. La utilización de punteros implica tomar precauciones de este tipo, ya que, como el lector recordará, el puntero siempre apunta a un espacio determinado de la memoria. Por lo tanto, es imprescindible hacer esta reserva de memoria para que nuestro programa no lleve a cabo un funcionamiento erróneo.

La otra función utilizada en la sentencia que estamos analizando es `sizeof()`, y como su nombre indica, retorna el tamaño que ocupa en memoria el tipo de dato pasado por argumento, y lo hace durante la ejecución del programa. En este caso, ya que estamos haciendo una petición de reserva de memoria para un puntero a `tipoNodo`, parece obvio que se debe utilizar `sizeof()` para averiguar el tamaño de dicho tipo de dato.



**Ejemplo 7.2: Haciendo *pop* en nuestra pila para eliminar elementos**

Realizar una función denominada `pop()` que reciba como parámetro un puntero a un tipo Pila, de modo que dicha función sea capaz de eliminar el nodo ubicado en la parte superior de la pila, si lo hubiere

**Código**

```
int pop(Pila *pila)
{

    tipoNodo *nodo_aux;

    int int_aux;

    /* El nodo auxiliar apunta al nodo en la cima de la
        pila */
    nodo_aux = *pila;

    /* Si el nodo auxiliar de la cima es NULL, acabar
       función*/
    if (!nodo_aux)
    {
        return 0;
    }

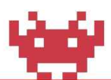
    /* Ahora, la pila apuntará al siguiente nodo al de
       la cima */
    *pila = nodo_aux->siguiente;

    /* Guardar valor del número del nodo a eliminar */
    int_aux = nodo_aux->numero;

    /* Libera la memoria del nodo a eliminar */
    free (nodo_aux);

    /* Retorna el valor del entero que había en el nodo
        eliminado */
    return int_aux;

}
```



**Ejemplo 7.3: Un ejemplo sencillo de inserción en pila**

Realizar un programa que utilice los tipos definidos anteriormente para poder declarar una pila e insertar consecutivamente un nodo con valor numérico 1, un nodo con valor numérico 2 y un nodo con valor numérico 3. Posteriormente, invocar a una función `imprimeElementos()` que muestre por la salida estándar los valores de los nodos de la pila, empezando por la parte superior de dicha pila y yendo hacia abajo.

**Código**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct nodo
{
    int numero;
    struct nodo *siguiente;
} tipoNodo;

typedef tipoNodo *Pila;

void push(Pila *pila, int valor);
int pop(Pila *pila);
void imprimeElementos(Pila *pila);

void main()
{
    Pila miPila = NULL;

    push(&miPila, 1);
    push(&miPila, 2);
    push(&miPila, 3);
    imprimeElementos(&miPila);
}


void push(Pila *pila, int valor)
{
    tipoNodo *nuevo;

    /* Crear un nodo nuevo */
    nuevo = (tipoNodo *)malloc(sizeof(tipoNodo));
```

```
    nuevo->numero = valor;

    /* El siguiente nodo al nuevo es el que antes era
        la cima de la pila */
    nuevo->siguiente = *pila;
    /* Ahora, nuestra pila apuntará al nuevo nodo */
    *pila = nuevo;
}


int pop(Pila *pila)
{
    tipoNodo *nodo_aux;
    int int_aux;

    /* El nodo auxiliar apunta al nodo en la cima de la
        pila */
    nodo_aux = *pila;

    /* Si el nodo auxiliar de la cima es NULL, acabar
        función*/
    if (!nodo_aux)
    {
            return 0;
    }

    /* Ahora, la pila apuntará al siguiente nodo al de
        la cima */
    *pila = nodo_aux->siguiente;

    /* Guardar valor del número del nodo a eliminar */
    int_aux = nodo_aux->numero;

    /* Libera la memoria del nodo a eliminar */
    free (nodo_aux);

    /* Retorna el valor del entero que había en el nodo
        eliminado */
    return int_aux;
}

void imprimeElementos(Pila *pila)
{

```

```
    tipoNodo *nodo_aux;
    int contador = 0;

    /** Mientras el nodo actual no sea NULO, imprimir
        su valor y avanzar en la pila **/
    while (nodo_aux)
    {
        contador++;
        printf ("Elemento %i: %i \n", contador, nodo_aux
            ->numero);
        nodo_aux = nodo_aux->siguiente;
    }

}
```

## 7.3. Las torres de Hanoi

Cuenta una antigua leyenda que hace muchos años, en el sagrado templo de Benarés, a orillas del río Ganges, se encontraba el epicentro del mundo. Aquel punto venía marcado por una bóveda bajo la que descansaba una base de bronce en la que se acomodaban tres agujas fabricadas con diamante. Cada aguja tenía el ancho del cuerpo de una abeja y se levantaba del suelo una altura de medio metro. Cuando Dios creó el mundo, utilizó una de estas agujas para colocar 64 discos de oro, cada uno con un diámetro distinto, de forma que el más grande sostenía al resto, disponiéndose hacia arriba de mayor a menor tamaño hasta alcanzar la cima, en la que se situaba el disco más pequeño.

Dios estableció una serie de normas que los sacerdotes del templo debían seguir a rajatabla para mover los discos de una aguja a otra. Dichas reglas se resumían en dos: primero, no podían mover más de un disco a la vez. Segundo, a la hora de insertar un disco en una aguja, dicho disco no puede tener un diámetro mayor que el que se encuentra en ese momento en la cima de la aguja. Tercero, solo se puede mover el disco que se encuentre en la cima de la aguja escogida en cada caso. Así, a través de un complejo procedimiento, los 64 discos debían quedar ensartados en la tercera aguja, respetando el orden que presentaban en la primera de ellas. Cuando alcanzaran su objetivo, el templo, los sacerdotes y el resto del mundo se transformarían en polvo, alcanzándose el fin de los días tal y como hoy los conocemos.

Esta leyenda fue inventada en 1883 por un matemático francés llamado Édouard Lucas d'Amiens para dotar de mayor atractivo y relevancia al problema de las torres y también a su invención. En aquella época, tal mecanismo publicitario era habitual en el gremio. Sin embargo, resulta interesante pararse a pensar por un momento en la leyenda. Si fuera cierta, ¿cuánto tardarían los sacerdotes en completar el puzzle y,

por lo tanto, fijar la fecha del Apocalipsis? La respuesta puede calcularse al elevar 2 a una potencia igual al número de discos, restándole una unidad a dicho resultado. Es decir, en este caso, 2 elevado a 64 - 1 daría un total de 18446744073709551615 movimientos. Si establecemos una velocidad de resolución de un movimiento por segundo, estaríamos hablando de un tiempo total de 585 mil millones de años.

A la vista de los hechos, tenemos tiempo de sobra antes de que acabe el mundo para planteaos un ejercicio algo más modesto que el que ideó Lucas d'Amiens. Partiremos de la base de las tres agujas de Hanoi, pero únicamente moveremos tres discos. El programa planteará dicho problema al jugador, y le permitirá hacer movimientos de discos entre las torres, siempre respetando las reglas que se establecieron en su momento. El programa mostrará de forma visual la situación actual de los discos y las agujas. Para ello, imprimirá el contenido de cada torre, representando cada disco con un número, siendo 1 el de diámetro más reducido y 3 el más grande. Cuando el jugador logre colocar los tres discos en la última aguja de manera ordenada, el programa llegará a su fin, informando al jugador del número de movimientos que ha necesitado para alcanzar su objetivo.

Como pistas diremos que lo ideal es programar varias funciones que irán siendo invocadas en el flujo del programa. Aparte de las funciones ya conocidas de `push()` y `pop()` para la pila, definiremos una función `imprimeDiscos()` que reciba una pila y muestre por pantalla los elementos que contiene. Además, será de utilidad definir una función `compruebaAlambreVacio()` que reciba una pila y sea capaz de discernir si dicha pila tiene o no elementos. Otra función podría ser `compruebaMovimientoLegal()` que reciba una pila origen y otra pila destino, realizando las comprobaciones necesarias para determinar si es posible mover un disco de origen a destino. Finalmente, definiremos una función `compruebaExito()` que recibirá la pila que representa a la tercera aguja. En el cuerpo de la función se comprobará si dicha pila tiene los tres discos colocados de manera ordenada, para saber si el flujo del programa ha llegado a la condición final.



**Juego 7.1: El juego de las torres de Hanoi**

Programar el código que implemente los requisitos necesarios para que el usuario pueda jugar al videojuego de las torres de Hanoi. El jugador podrá, en cada turno, escoger el movimiento que desee llevar a cabo, especificando la torre origen desde la que mueve el disco y la torre destino a la que quiere llevar dicho disco. Al final de cada turno, representar el estado de las tres torres. Avisar al jugador cuando haya completado el videojuego.

**Código**

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

typedef struct nodo
{
   int numero;
   struct nodo *siguiente;
}tipoNodo;

typedef tipoNodo *Pila;

void push(Pila *pila, int valor);
int pop(Pila *pila);
void imprimeDiscos(int numero, Pila *pila);
int compruebaAlambreVacio (Pila *pila);
int compruebaMovimientoLegal (Pila *pilaOrigen, Pila *
    pilaDestino);
void movimientoIlegal();
int compruebaExito (Pila *pila, int numDiscos);
void limpiapantalla();

void main()
{
   /** Variables de control **/
   int movimiento;
   int valorExtraido;
   int intentos = 0;
   int movimientoIlegal = 0;

   /** Número de discos **/
   int numDiscos = 3;

   /** Inicializa alambres **/
   Pila alambre1 = NULL;
   Pila alambre2 = NULL;
   Pila alambre3 = NULL;

   push(&alambre1, 3);
   push(&alambre1, 2);
   push(&alambre1, 1);

   do
   {
      /** Borra pantalla **/
      limpiapantalla();
```

```c

        /** Imprime información del último movimiento realizado **/
        if (movimientoIlegal == 1)
        {
            printf("\n El movimiento que has seleccionado es ilegal \n");
            movimientoIlegal = 0;
        }

        /** Imprime alambre por pantalla **/
        imprimeDiscos(1, &alambre1);
        imprimeDiscos(2, &alambre2);
        imprimeDiscos(3, &alambre3);

        /** Presenta opciones **/
        puts("------------------------------------");
        puts("1 - Mueve de alambre 1 a alambre 2");
        puts("2 - Mueve de alambre 1 a alambre 3");
        puts("3 - Mueve de alambre 2 a alambre 1");
        puts("4 - Mueve de alambre 2 a alambre 3");
        puts("5 - Mueve de alambre 3 a alambre 1");
        puts("6 - Mueve de alambre 3 a alambre 2");
        printf("Llevas %i movimientos hasta el momento \n", intentos);
        puts("Escoge tu siguiente movimiento introduciendo el número");
        scanf("%i", &movimiento);

        switch (movimiento)
        {
            case 1:
            if (compruebaMovimientoLegal(&alambre1, &alambre2))
            {
                valorExtraido = pop(&alambre1);
                push(&alambre2, valorExtraido);
                intentos++;
            }
            else
            {
                movimientoIlegal = 1;
            }
            break;
```

```
case 2:
if (compruebaMovimientoLegal(&alambre1, &alambre3))
{
    valorExtraido = pop(&alambre1);
    push(&alambre3, valorExtraido);
    intentos++;
}
else
{
    movimientoIlegal = 1;
}
break;
case 3:
if (compruebaMovimientoLegal(&alambre2, &alambre1))
{
    valorExtraido = pop(&alambre2);
    push(&alambre1, valorExtraido);
    intentos++;
}
else
{
    movimientoIlegal = 1;
}
break;
case 4:
if (compruebaMovimientoLegal(&alambre2, &alambre3))
{
    valorExtraido = pop(&alambre2);
    push(&alambre3, valorExtraido);
    intentos++;
}
else
{
    movimientoIlegal = 1;
}
break;
case 5:
if (compruebaMovimientoLegal(&alambre3, &alambre1))
{
    valorExtraido = pop(&alambre3);
```

```
                push(&alambre1, valorExtraido);
                intentos++;
            }
            else
            {
                movimientoIlegal = 1;
            }
            break;
            case 6:
            if (compruebaMovimientoLegal(&alambre3, &alambre2))
            {
                valorExtraido = pop(&alambre3);
                push(&alambre2, valorExtraido);
                intentos++;
            }
            else
            {
                movimientoIlegal = 1;
            }
            break;
            default:
            break;
        }

    }
    while (!(compruebaExito(&alambre3, numDiscos)));

    limpiapantalla();
    imprimeDiscos(1, &alambre1);
    imprimeDiscos(2, &alambre2);
    imprimeDiscos(3, &alambre3);

    printf("\n Has terminado el juego en %i movimientos", intentos);
}


void push(Pila *pila, int valor)
{
    tipoNodo *nuevo;

    /* Crear un nodo nuevo */
    nuevo = (tipoNodo *)malloc(sizeof(tipoNodo));
```

```
    nuevo->numero = valor;

    /* El siguiente nodo al nuevo es el que antes era
        la cima de la pila */
    nuevo->siguiente = *pila;
    /* Ahora, nuestra pila apuntará al nuevo nodo */
    *pila = nuevo;
}


int pop(Pila *pila)
{
    tipoNodo *nodo_aux;
    int int_aux;

    /* El nodo auxiliar apunta al nodo en la cima de la
         pila */
    nodo_aux = *pila;

    /* Si el nodo auxiliar de la cima es NULL, acabar
        función*/
    if (!nodo_aux)
    {
        return 0;
    }

    /* Ahora, la pila apuntará al siguiente nodo al de
        la cima */
    *pila = nodo_aux->siguiente;

    /* Guardar valor del número del nodo a eliminar */
    int_aux = nodo_aux->numero;

    /* Libera la memoria del nodo a eliminar */
    free (nodo_aux);

    /* Retorna el valor del entero que había en el nodo
         eliminado */
    return int_aux;
}

void imprimeDiscos(int numero, Pila *pila)
{

```

```
    tipoNodo *nodo_aux;
    nodo_aux = *pila;

    printf("\n Alambre num. %i \n", numero);

    if (!(nodo_aux))
    {
        printf ("Sin discos \n");
    }
    else
    {
        /** Mientras el nodo actual no sea NULO,
            imprimir su valor y avanzar en la pila **/
        while (nodo_aux)
        {
            printf ("%i \n", nodo_aux->numero);
            nodo_aux = nodo_aux->siguiente;
        }
    }
}

int compruebaMovimientoLegal (Pila *pilaOrigen, Pila *
    pilaDestino)
{

    int valor_origen;
    int valor_destino;
    tipoNodo *nodo_aux_origen;
    tipoNodo *nodo_aux_destino;

    /** Comprobar si el valor origen puede colocarse
        sobre el valor destino **/
    /** Para ello, el valor origen debe ser menor que
        el destino **/

    /** Si la pila origen está vacía, movimiento ilegal
        **/
    if (!(*pilaOrigen))
    {
        return 0;
    }

    nodo_aux_origen = *pilaOrigen;
    valor_origen = nodo_aux_origen->numero;
```

```

    /** Comprobar si la pila destino está vacía **/
    if (*pilaDestino)
    {
        nodo_aux_destino = *pilaDestino;
        valor_destino = nodo_aux_destino->numero;

        /** Si el disco origen es más grande que el
            destino, ilegal **/
        if (valor_origen > valor_destino)
        {
            return 0;
        }
    }

    /** Si llega hasta aquí, el movimiento es legal **/
    return 1;
}

int compruebaExito (Pila *pila, int numDiscos)
{

    int contador = 0;
    int valor_actual;
    tipoNodo *nodo_aux;

    /** Comprobar la condición de éxito **/
    /** Los elementos en la pila siempre irán de menor
        a mayor y alcanzan el número total de discos *
        */
    nodo_aux = *pila;

    if (!(nodo_aux))
    {
        return 0;
    }

    while(contador < numDiscos)
    {
        if (nodo_aux)
        {
            contador++;
            valor_actual = nodo_aux->numero;
            nodo_aux = nodo_aux->siguiente;
```

```
                if (nodo_aux && (nodo_aux->numero <
                    valor_actual))
                {
                    return 0;
                }
            }
            else
            {
                return 0;
            }
        }

        return 1;
}

void limpiapantalla()
{
    int i;
    for(i=1; i<=100; i++)
        printf("\n");
}
```

## 7.4. Ejercicios propuestos

Llegamos al final de la obra, pero reprimid aún vuestras lágrimas. Todavía no hemos terminado. Queda vuestra parte favorita del capítulo: los ejercicios propuestos. Ya que este es un capítulo de ampliación, seremos buenos y concluiremos con un par de ejercicios sencillos antes de poner punto y final.



**Ejercicio 7.1: Malos apilados**

Se quiere programar un un matamarcianos en el que aparezcan naves enemigas por el extremo de la pantalla. Dichas naves pueden ser destruidas si son alcanzadas por los disparos de la nave del protagonista. Puede asumirse que pueden ser necesarios varios disparos para derribar una nave, ya que hay enemigos más fuertes que otros. ¿Qué datos debería contener una estructura de datos utilizada para las naves enemigas? ¿Qué funciones deberían estar presentes en un juego de estas características?



**Ejercicio 7.2: ¿Dónde está mi memoria?**

En este capítulo hemos realizado reservas de memoria dinámica. En principio, hemos asumido que la función `malloc` siempre realiza la reserva de memora con éxito. Sin embargo, si el ordenador no tiene memoria disponible la reserva no se realiza y la función devuelve NULL. Modifica los programas realizados para tener en cuenta este supuesto.

Si habéis llegado hasta aquí con éxito, podéis escribir a Hogwarts para que os envíen el título de "Mago de la Programación". ¡Enhorabuena! Por si aún no sois conscientes, habéis adquirido una serie de habilidades muy útiles en este libro. Esperamos que os abran muchas puertas y que continuéis por este sendero tal y como hicimos nosotros hace ya unos años. La dedicación y el amor por lo que se hace os ayudarán a conservar y a ampliar lo aprendido, así que nunca dejéis de programar, nunca dejéis de jugar. Al fin y al cabo, a programar se aprende jugando.

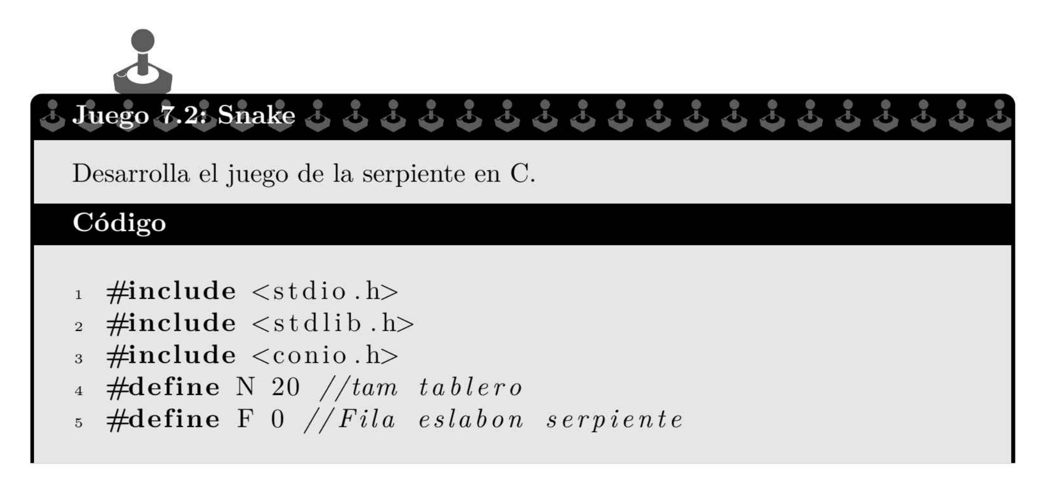
      

# Fase de bonus: juego de la serpiente

Todo juego que se precie contiene una fase extra para alegría de los jugadores. En esta obra no queremos ser menos y hemos dejado para el final un juego que seguro que os gustará: la serpiente. Aunque fue programado en los años 70, el juego no alcanzó fama mundial hasta finales de los 90 del siglo pasado, cuando Nokia lo incluyó preinstalado en su gama de teléfonos móviles.

Se trata de un juego tan sencillo como adictivo: el jugador controla a una hambrienta serpiente que debe comer todo lo que pueda para crecer. El desafío radica en que la serpiente se encuentra siempre en movimiento dentro de un escenario cerrado; si "choca" contra las paredes del recinto o contra sí misma, el juego termina. Así pues, se trata de conseguir crecer tanto como sea posible antes de que el juego termine, ya que tarde o temprano la serpiente dejará de caber en el espacio que la contiene. Es difícil conseguir más con menos, y ahí radica el encanto de este juego.

Antes de continuar, debemos advertir que este juego no se ha introducido en ninguno de los capítulos anteriores porque utiliza una librería llamada `conio.h`, que solo está disponible para Windows. Dicha librería contiene la función `kbhit`, que es muy útil en este tipo de aplicaciones. Esta función permite saber si se ha pulsado el teclado, lo que evita paralizar el juego mientras se espera a que el usuario realice acciones con el teclado. Existen algunas alternativas para Linux o Mac, pero van más allá del alcance del libro.



**Juego 7.2: Snake**

Desarrolla el juego de la serpiente en C.

**Código**

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define N 20 //tam tablero
#define F 0 //Fila eslabon serpiente
```

```
#define C 1 //columna eslabon serpiente
#define FRUTA -1
#define BORDE 5
#define PROB_PREMIO

int main()
{
    char c='d'; //para que avance a la derecha
        inicialmente
    int tablero[N][N];
    int serpiente[N*N][2];
    int i, j, ih=1, it=0, ipremio, jpremio;
    int vivo=1; // flag para seguir jugando

    // Inicio generador numeros aleatorios
    srand(time(NULL));

    // Inicializo el tablero
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
        {
            if(i==0||i==N-1||j==0||j==N-1)
                tablero[i][j]=BORDE;
            else
                tablero[i][j]=0;
        }

    // La serpiente comienza en N/2, N/2
    serpiente[it][F]=N/2;
    serpiente[it][C]=N/2;
    serpiente[ih][F]=N/2;
    serpiente[ih][C]=N/2+1;

    // bucle del juego
    while(vivo)
    {

        // Generacion de premios
        if((rand()%100)<PROB_PREMIO)
        {
            //premio DENTRO del recinto
            ipremio=rand()%(N-1)+1;
            jpremio=rand()%(N-1)+1;
            tablero[ipremio][jpremio]=FRUTA;
```

```
        }

        // Si se ha pulsado el teclado...
        if(kbhit())
            c=getch();

        // Borro la cola de la serpiente
        tablero[serpiente[it][F]][serpiente[it][C]]=0;

        // Actualizo el cuerpo (la cabeza NO)
        for(i=it;i<ih;i++)
        {
            serpiente[i][F]=serpiente[i+1][F];
            serpiente[i][C]=serpiente[i+1][C];
        }


        // Actualizo la cabeza
        switch(c)
        {
            case 'w': //arriba
            serpiente[ih][F]--;
            break;

            case 's': //abajo
            serpiente[ih][F]++;
            break;

            case 'a': //izquierda
            serpiente[ih][C]--;
            break;

            case 'd': //derecha
            serpiente[ih][C]++;
            break;
        }


        //Actualizo cabeza y cola segun donde vaya a aterizar la cabeza
        switch(tablero[serpiente[ih][F]][serpiente[ih][C]])
        {
            case 0:  //no hay problema
```

```
                break;

            case FRUTA:     //has comido (se aumenta la longitud)
                ih++;
                serpiente[ih][F]=serpiente[ih-1][F];
                serpiente[ih][C]=serpiente[ih-1][C];
                break;

            default:    //te has mordido
                vivo=0;
                break;
        }


        //Actualizo el tablero
        for(i=it;i<=ih;i++)
        tablero[serpiente[i][F]][serpiente[i][C]]=i+1;


        //Dibujo el tablero
        for(i=0; i<N; i++)
        {
            for(j=0; j<N; j++)
            {
                if(tablero[i][j]>0)
                    printf("*");
                else if (tablero[i][j]==FRUTA)
                    printf("+");
                else
                    printf(" ");
            }
            printf("\n");
        }
        usleep(200000);   //ralentiza la representacion
        system("cls");    //borra la pantalla
    }

    printf("Tu serpiente ha llegado a medir %d metros !!!", ih+1);
}
```

La programación del juego es sencilla: se utiliza una matriz de tamaño N×N llamada `tablero` como recinto para la serpiente. Cualquier elemento de la matriz mayor

que 0 es considerado como obstáculo (la propia serpiente y los bordes) y es representado con un asterisco en pantalla. Los trozos de comida son representados con el caracter '+', aparecen aleatoriamente y se les asigna un -1 al elemento de la matriz que los contiene. El resto de elementos de la matriz contiene 0 y su representación en pantalla es un espacio en blanco.

La gestión de la serpiente merece comentario aparte por la manera en la que se realiza: se utiliza una matriz llamada `serpiente` con N×N filas y dos columnas. En cada fila se puede almacenar la posición de un eslabón de la serpiente, es decir, las coordenadas de dicho eslabón dentro de la matriz `tablero` (la primera columna almacena la fila y la segunda la columna). Dado que la longitud de la serpiente cambia durante el juego, se utilizan dos variables auxiliares, `ih`, que guarda la fila de `serpiente` donde están las coordenadas de la cabeza de la serpiente, e `it`, que hace lo propio con las coordenadas de la cola. Es decir, las coordenadas de cada uno de los elementos que componen a la serpiente se encuentran entre las filas `ih` e `it` de la matriz `serpiente`. Esta matriz se actualiza dinámicamente a medida que se mueve o crece la serpiente.

La serpiente se controla por el teclado (w: arriba, a: izquierda, s: abajo y d: derecha). La representación gráfica utiliza dos funciones auxiliares que no hemos visto anteriormente: `system("cls")`, que limpia la pantalla, y `usleep(200000)`, que "congela" el programa durante 200000 microsegundos para que la velocidad a la que se ejecuta el juego sea compatible con el disfrute del jugador. (¡El ordenador es mucho más rápido que nosotros!) Esta cantidad puede ser ajustada al gusto, por ejemplo para aumentar la velocidad del juego.

Finalmente, y como no podía ser de otra manera, la fase bonus termina con un reto de programación.



**Ejercicio 7.3: Serpiente con funciones**

Desarrolla el juego de la serpiente utilizando funciones.

Programar consiste en dar órdenes a una máquina para que las ejecute de forma automática. Exige, únicamente, hablar un idioma que la máquina entienda: un lenguaje de programación. Con las herramientas apropiadas, basta escribir unas decenas de palabras para que nuestros deseos cobren vida en los circuitos de un ordenador. Es magia. Y está al alcance de todo aquel que esté dispuesto a invertir unas horas de su tiempo en estas páginas.

El lenguaje de programación que el usuario aprenderá con este libro es C, el clásico por antonomasia. Si la música de los Beatles o los Rolling Stones se convirtiera en código, lo haría en C, sin duda. A diferencia de otros textos, en este se ha apostado por la programación de videojuegos como vehículo principal de aprendizaje. ¿Qué mejor forma de aprender que con ejemplos inspirados por clásicos como Hundir la Flota, Trivial Pursuit, Monkey Island o Angry Birds?

A programar se tiene que aprender programando, sí, pero también jugando.

**José M.ª Maestre Torreblanca** (Sevilla, 1982) es doctor ingeniero de Telecomunicaciones y profesor del Departamento de Ingeniería de Sistemas y Automática de la Universidad de Sevilla. Amante de los videojuegos desde la infancia, fue colaborador de *MeriStation* durante un año. Además, es autor y coautor de más de un centenar de publicaciones científicas, incluidos los libros *Service Robotics within the Digital Home* (Springer, 2011), *Distributed Model Predictive Control Made Easy* (Springer, 2014) y *Domótica para ingenieros* (Paraninfo, 2015). En la actualidad, reside en Japón, merced a un proyecto conjunto con el Departamento de Informática del Instituto de Tecnología de Tokio.

**Jesús Relinque Pérez** (Cádiz, 1980) es ingeniero técnico en Informática. Tras colaborar con el medio digital de videojuegos *MeriStation* durante cinco años, creó la web *PixeBlog de Pedja* en 2006, donde realizó una labor de arqueología a través de la historia del videojuego. En 2015 ahondó en dicha cuestión con la publicación del libro *Génesis: guía esencial de los videojuegos españoles de 8bits* (Héroes de Papel, 2015). En 2016 participó en *Conectados*, programa radiofónico de Canal Sur Radio, así como en la revista especializada *Retro Gamer*.





